Types of Classes in C#

Introduction of C#
As we know, C# is an Object Oriented Programming language that provides the ability to reuse existing code. To reuse existing code, C# provides various object-oriented concepts such as classes, objects, properties, methods, structs, and records.

What is a class in C#?
Classes are user-defined data types that represent the state and behavior of an object. The state represents the properties, and behavior is the action that objects can perform.

Classes can be declared using the following access specifiers that limit the accessibility of classes to other classes. However, some classes do not require any access modifiers.

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

For example

```
public class Accounts
{

}
```

Some Key points about classes:
- Classes are reference types that hold the object created dynamically in a heap.
- All classes have a base type of System.Object.
- The default access modifier of a class is Internal.
- The default access modifier of methods and variables is Private.
- Directly inside the namespaces, declarations of private classes are not allowed.

Types of classes in C#

What is an Abstract Class in C#?
An abstract class is a class that provides a common definition to the subclasses, and this is the type of class whose object is not created.

Some key points of Abstract classes are:
- Abstract classes are declared using the abstract keyword.
- We cannot create an object of an abstract class.
- It must be inherited in a subclass if you want to use it.
- An Abstract class contains both abstract and non-abstract methods.
- The methods inside the abstract class can either have an or no implementation.

- We can inherit two abstract classes; in this case, implementation of the base class method is optional.
- An Abstract class has only one subclass.
- Methods inside the abstract class cannot be private.
- If there is at least one method abstract in a class, then the class must be abstract.

For example

```
abstract class Accounts
{

}
```

What are Partial Classes in C#

It is a type of class that allows dividing their properties, methods, and events into multiple source files, and at compile time, these files are combined into a single class.

The following are some key points:

- All the parts of the partial class must be prefixed with the partial keyword.
- If you seal a specific part of a partial class, the entire class is sealed, the same as for an abstract class.
- Inheritance cannot be applied to partial classes.
- The classes written in two class files are combined at run time.

For example

```
partial class Accounts
{

}
```

What is Sealed Class in C#?

A Sealed class is a class that cannot be inherited and used to restrict the properties.

The following are some key points:

- A Sealed class is created using the sealed keyword.
- Access modifiers are not applied to a sealed class.
- To access the sealed members, we must create an object of the class.

For example

```
sealed class Accounts
{

}
```

What is Static Class in C#?
It is the type of class that cannot be instantiated. In other words, we cannot create an object of that class using the new keyword, such that class members can be called directly using their name.

The following are some key points:

- It was created using the static keyword.
- Only static members are allowed; in other words, everything inside the class must be static.
- We cannot create an object of the static class.
- A Static class cannot be inherited.
- It allows only a static constructor to be declared.
- The static class methods can be called using the class name without creating the instance.

For example

```
static class Accounts
{

}
```

Generics

In C#, a generic class is a class that can work with any data type, allowing you to define classes with members whose data types are specified when the class is instantiated. Here's a simple example of a generic class in C#:

```
using System;

public class MyGenericClass<T>
{
    private T myData;

    public MyGenericClass(T data)
    {
        myData = data;
    }

    public void DisplayData()
    {
        Console.WriteLine($"Data: {myData}");
    }
}

class Program
{
    static void Main()
    {
        // Create an instance of MyGenericClass with int data type
        MyGenericClass<int> intInstance = new MyGenericClass<int>(42);
        intInstance.DisplayData();

        // Create an instance of MyGenericClass with string data type
        MyGenericClass<string> stringInstance = new MyGenericClass<string>("Hello,
Generics!");
        stringInstance.DisplayData();
    }
}
```

In this example, MyGenericClass<T> is a generic class where T is a placeholder for the actual data type. The constructor and DisplayData method can work with any data type specified when creating an instance of the class. The Main method demonstrates how to create instances of MyGenericClass with different data types (int and string in this case).

Generics are particularly useful for creating reusable and type-safe code, as they allow you to write classes and methods that can work with various data types without sacrificing type safety.

File System

In C#, the .NET Framework provides a rich set of libraries for working with file systems. The primary namespace for file system operations is System.IO. Here's an overview of some important classes and methods for file system operations in C#:

1. **System.IO.File Class:**
- File.Exists(string path): Checks if a file exists at the specified path.
- File.WriteAllText(string path, string contents): Writes the specified text content to a file, creating the file if it doesn't exist.
- File.ReadAllLines(string path): Reads all lines from a file into a string array.

2. **System.IO.Directory Class:**
- Directory.Exists(string path): Checks if a directory exists at the specified path.
- Directory.CreateDirectory(string path): Creates a new directory at the specified path.
- Directory.GetFiles(string path): Returns an array of file names in the specified directory.
- Directory.GetDirectories(string path): Returns an array of directory names in the specified directory.

3. **System.IO.Path Class:**
- Path.Combine(string path1, string path2): Combines two strings into a path.
- Path.GetExtension(string path): Gets the extension of the specified path.
- Path.GetFileName(string path): Gets the file name and extension from a path.

4. **File and Directory Operations:**
- Reading/Writing Bytes: Use File.ReadAllBytes and File.WriteAllBytes for reading and writing binary data.
- Copying Files/Directories: File.Copy and Directory.Copy methods can be used for copying files and directories.
- Moving/Renaming: File.Move and Directory.Move methods are used for moving or renaming files and directories.

5. **System.IO.Stream Class:**
- For low-level file I/O operations, you can use streams. FileStream is commonly used for reading and writing bytes to files.

6. **Handling Exceptions:**
- Always handle exceptions that may occur during file system operations, especially when dealing with user inputs or external files.

Here's a simple example that demonstrates some basic file system operations:

```
using System;
using System.IO;

class Program
```

```
{
    static void Main()
    {
        // Example file path
        string filePath = "example.txt";

        // Check if the file exists
        if (File.Exists(filePath))
        {
            // Read all lines from the file
            string[] lines = File.ReadAllLines(filePath);

            // Display each line
            foreach (string line in lines)
            {
                Console.WriteLine(line);
            }
        }
        else
        {
            Console.WriteLine($"File '{filePath}' does not exist.");
        }

        // Example directory path
        string directoryPath = "ExampleDirectory";

        // Check if the directory exists
        if (!Directory.Exists(directoryPath))
        {
            // Create the directory
            Directory.CreateDirectory(directoryPath);
            Console.WriteLine($"Directory '{directoryPath}' created.");
        }
        else
        {
            Console.WriteLine($"Directory '{directoryPath}' already exists.");
        }
    }
}
```

Remember to handle exceptions appropriately in a production environment, and consider security implications when working with file paths and user inputs.

Data Serialization

Serialization is the process of converting an object or data structure into a format that can be easily stored or transmitted and later reconstructed. In C#, there are several ways to perform data serialization. Two common methods are using the built-in BinaryFormatter or XML serialization. Here's an overview of both approaches:

1. **Binary Serialization (using BinaryFormatter):**

Binary serialization is a compact and efficient way to serialize objects into a binary format. The BinaryFormatter class in the System.Runtime.Serialization.Formatters.Binary namespace is commonly used for this purpose.

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
public class MyClass
{
   public int MyProperty { get; set; }
   public string AnotherProperty { get; set; }
}

class Program
{
   static void Main()
   {
      // Create an instance of MyClass
      MyClass myObject = new MyClass
      {
         MyProperty = 42,
         AnotherProperty = "Hello, Serialization!"
      };

      // Binary serialization
      BinaryFormatter formatter = new BinaryFormatter();
      using (FileStream stream = new FileStream("data.dat", FileMode.Create))
      {
         formatter.Serialize(stream, myObject);
      }

      // Binary deserialization
      using (FileStream stream = new FileStream("data.dat", FileMode.Open))
      {
         MyClass deserializedObject = (MyClass)formatter.Deserialize(stream);
```

```
            Console.WriteLine(deserializedObject.MyProperty);  // Output: 42
            Console.WriteLine(deserializedObject.AnotherProperty);  // Output: Hello,
Serialization!
        }
    }
}
```

2. **XML Serialization (using XmlSerializer):**

XML serialization is often used when human-readable data is desirable. The XmlSerializer class in the System.Xml.Serialization namespace is used for XML serialization.

```
using System;
using System.IO;
using System.Xml.Serialization;

[Serializable]
public class MyClass
{
    public int MyProperty { get; set; }
    public string AnotherProperty { get; set; }
}

class Program
{
    static void Main()
    {
        // Create an instance of MyClass
        MyClass myObject = new MyClass
        {
            MyProperty = 42,
            AnotherProperty = "Hello, Serialization!"
        };

        // XML serialization
        XmlSerializer serializer = new XmlSerializer(typeof(MyClass));
        using (TextWriter writer = new StreamWriter("data.xml"))
        {
            serializer.Serialize(writer, myObject);
        }

        // XML deserialization
        using (TextReader reader = new StreamReader("data.xml"))
        {
            MyClass deserializedObject = (MyClass)serializer.Deserialize(reader);
            Console.WriteLine(deserializedObject.MyProperty);  // Output: 42
            Console.WriteLine(deserializedObject.AnotherProperty);  // Output: Hello,
Serialization!
```

```
        }
    }
}
```

3. **Json Serialization (using Newtonsoft.Json)**

The Newtonsoft.Json library (Json.NET) is a popular third-party library for JSON serialization and deserialization in C#. You can install it using NuGet Package Manager with the following command:

```
Install-Package Newtonsoft.Json
```

Once you have it installed, you can use the JsonConvert class from the Newtonsoft.Json namespace for serialization and deserialization. Here's an example:

```
using System;
using System.IO;
using Newtonsoft.Json;

public class MyClass
{
    public int MyProperty { get; set; }
    public string AnotherProperty { get; set; }
}

class Program
{
    static void Main()
    {
        // Create an instance of MyClass
        MyClass myObject = new MyClass
        {
            MyProperty = 42,
            AnotherProperty = "Hello, Newtonsoft.Json Serialization!"
        };

        // JSON serialization
        string jsonString = JsonConvert.SerializeObject(myObject);
        File.WriteAllText("data.json", jsonString);

        // JSON deserialization
        string jsonFromFile = File.ReadAllText("data.json");
        MyClass deserializedObject =
JsonConvert.DeserializeObject<MyClass>(jsonFromFile);

        Console.WriteLine(deserializedObject.MyProperty);  // Output: 42
        Console.WriteLine(deserializedObject.AnotherProperty);  // Output: Hello,
Newtonsoft.Json Serialization!
```

```
        }
    }
```

In this example:

- JsonConvert.SerializeObject is used to convert the myObject instance into a JSON-formatted string.
- File.WriteAllText is used to write the JSON string to a file.
- File.ReadAllText is used to read the JSON string from the file.
- JsonConvert.DeserializeObject<MyClass> is used to convert the JSON string back into a MyClass object.

Json.NET provides various customization options and features for handling JSON data, making it a powerful and flexible choice for JSON serialization in C#.

Binary Serialization :-

Binary serialization in C# is the process of converting an object into a stream of bytes. This stream can then be stored in a file or transmitted over a network. When the stream is read back into memory, it can be converted into an object again.

Binary serialization preserves type fidelity, which means that the complete state of the object is recorded and when you deserialize, an exact copy is created. This type of serialization is useful for preserving the state of an object between different invocations of an application.

Here are some general steps for serialization in C#:

- Create an instance of a File that will store serialized objects.
- Create a stream from the file object.
- Create an instance of BinaryFormatter.
- Call the serialize method of the instance, passing it stream and object to serialize.

Binary serialization can be done through the BinaryFormatter class located inside the Mscorlib assembly. For this, you need to add the System.Runtime.Serialization

Base Library Features

C# (C Sharp) is a modern, object-oriented programming language developed by Microsoft. The base class library (BCL) in C# provides a set of fundamental classes and types that are essential for building .NET applications. Here are some key features and components of the base class library in C#:

1. **System Namespace:**
- The System namespace is a fundamental namespace that contains essential types such as Object, String, Exception, etc.
- The System namespace also includes fundamental numeric types like Int32, Double, Decimal, etc.

2. **Collections:**
- The System.Collections namespace provides classes for working with collections such as arrays, lists, queues, and dictionaries. Some important classes include ArrayList, List<T>, Queue, Stack, Dictionary<TKey, TValue>, etc.

3. **IO (Input/Output):**
- The System.IO namespace includes classes for file and stream operations, allowing you to read from and write to files. Key classes include File, FileStream, StreamReader, and StreamWriter.

4. **Threading:**
- The System.Threading namespace provides classes for multi-threading and asynchronous programming. Key classes include Thread, ThreadPool, and Task.

5. **Reflection:**
- The System.Reflection namespace allows you to inspect and interact with metadata of types, assemblies, and modules at runtime. Key classes include Assembly, Type, and MethodInfo.

6. **Networking:**
- The System.Net namespace provides classes for networking tasks, including classes for working with sockets (Socket class), web clients (WebClient class), and more.

7. **LINQ (Language Integrated Query):**
- The System.Linq namespace is a part of the Language Integrated Query (LINQ) feature in C#. It provides classes and interfaces for querying collections and other data sources using a SQL-like syntax.

8. **XML and Serialization:**
- The System.Xml namespace provides classes for working with XML, including XmlDocument, XmlElement, and XML serialization classes like XmlSerializer.

9. **Attributes:**
- The System.Attributes namespace provides classes for defining custom attributes, which can be used to add metadata to types and members.

10. **Security:**
- The System.Security namespace contains classes for implementing security features, including cryptography and access control.

These are just a few examples of the many features provided by the base class library in C#. The BCL is an integral part of the .NET Framework (or .NET Core/.NET 5 and later) and is crucial for building a wide range of applications, from desktop to web and mobile applications.

Lambda Expression

Lambda expressions in C# are used like anonymous functions, with the difference that in Lambda expressions you don’t need to specify the type of the value that you input thus making it more flexible to use.

The ‘=>’ is the lambda operator which is used in all lambda expressions. The Lambda expression is divided into two parts, the left side is the input and the right is the expression.

The Lambda Expressions can be of two types:

1. **Expression Lambda**: Consists of the input and the expression.
   Syntax:
   input => expression;

2. **Statement Lambda:** Consists of the input and a set of statements to be executed.
   Syntax:
   input => { statements };

Let us take some examples to understand the above concept better.

Example 1: In the code given below, we have a list of integer numbers. The first lambda expression evaluates every element’s square { x => x*x } and the second is used to find which values are divisible by 3 { x => (x % 3) == 0 }. And the foreach loops are used for displaying.

```
// C# program to illustrate the
// Lambda Expression
using System;
using System.Collections.Generic;
using System.Linq;

namespace Lambda_Expressions {
class Program {
   static void Main(string[] args)
   {
      // List to store numbers
      List<int> numbers = new List<int>() {36, 71, 12,
                        15, 29, 18, 27, 17, 9, 34};

      // foreach loop to display the list
      Console.Write("The list : ");
      foreach(var value in numbers)
      {
         Console.Write("{0} ", value);
      }
      Console.WriteLine();
```

```
            // Using lambda expression
            // to calculate square of
            // each value in the list
            var square = numbers.Select(x => x * x);

            // foreach loop to display squares
            Console.Write("Squares : ");
            foreach(var value in square)
            {
                Console.Write("{0} ", value);
            }
            Console.WriteLine();

            // Using Lambda expression to
            // find all numbers in the list
            // divisible by 3
            List<int> divBy3 = numbers.FindAll(x => (x % 3) == 0);

            // foreach loop to display divBy3
            Console.Write("Numbers Divisible by 3 : ");
            foreach(var value in divBy3)
            {
                Console.Write("{0} ", value);
            }
            Console.WriteLine();
        }
    }
    }
```

Output:

```
The list : 36 71 12 15 29 18 27 17 9 34
Squares : 1296 5041 144 225 841 324 729 289 81 1156
Numbers Divisible by 3 : 36 12 15 18 27 9
```

Example 2: Lambda expressions can also be used with user-defined classes. The code given below shows how to sort through a list based on an attribute of the class that the list is defined upon.

```
// C# program to illustrate the
// Lambda Expression
using System;
using System.Collections.Generic;
using System.Linq;

// User defined class Student
class Student {
    // properties rollNo and name
    public int rollNo
    {
        get;
        set;
    }

    public string name
    {
        get;
        set;
    }
}

class GFG {

    // Main Method
    static void Main(string[] args)
    {
        // List with each element of type Student
        List<Student> details = new List<Student>() {
            new Student{ rollNo = 1, name = "Liza" },
                new Student{ rollNo = 2, name = "Stewart" },
                new Student{ rollNo = 3, name = "Tina" },
                new Student{ rollNo = 4, name = "Stefani" },
                new Student { rollNo = 5, name = "Trish" }
        };

        // To sort the details list
        // based on name of student
        // in ascending order
        var newDetails = details.OrderBy(x => x.name);

        foreach(var value in newDetails)
        {
            Console.WriteLine(value.rollNo + " " + value.name);
        }
    }
}
```

Output:

```
1 Liza
4 Stefani
2 Stewart
3 Tina
5 Trish
```

Extension Methods

In C#, extension methods allow you to add new methods to existing types without modifying them. These methods are defined in static classes and must be static themselves. Extension methods are an elegant way to enhance the functionality of existing types, including those in the .NET Framework or your own custom classes.

Here's an example of an extension method:

```
public static class StringExtensions
{
    public static string Reverse(this string input)
    {
        char[] charArray = input.ToCharArray();
        Array.Reverse(charArray);
        return new string(charArray);
    }
}
```

In this example, Reverse is an extension method for the string type. The this keyword before the first parameter indicates that it is an extension method for the string class.

To use this extension method:

```
class Program
{
    static void Main()
    {
        string original = "Hello, world!";
        string reversed = original.Reverse();

        Console.WriteLine(reversed);  // Output: "!dlrow ,olleH"
    }
}
```

When documenting extension methods, you should follow standard documentation practices. You can use XML comments to provide documentation that can be extracted by tools like IntelliSense or automatically generate documentation.

Here's an example of how you can document the extension method:

```
/// <summary>
/// Provides extension methods for the <see cref="System.String"/> class.
/// </summary>
public static class StringExtensions
{
    /// <summary>
    /// Reverses the characters in the given string.
    /// </summary>
    /// <param name="input">The input string to reverse.</param>
    /// <returns>A new string with characters reversed.</returns>
    public static string Reverse(this string input)
    {
        char[] charArray = input.ToCharArray();
        Array.Reverse(charArray);
        return new string(charArray);
    }
}
```

In this example:

- The <summary> tag provides a brief description of the purpose of the extension class.
- The <param> tag is used to document the parameter of the extension method.
- The <returns> tag is used to describe the return value of the extension method.

List Linq Operations

- **Projection Operators:**
    - Select: Projects each element of a sequence into a new form.
    - SelectMany: Projects each sequence element to an IEnumerable<T> and flattens the resulting sequences into one sequence.

- **Restriction Operators:**
    - Where: Filters a sequence of values based on a predicate.

- **Partitioning Operators:**
    - Take: Returns a specified number of contiguous elements from the start of a sequence.
    - Skip: Bypasses a specified number of elements in a sequence and then returns the remaining elements.
    - TakeWhile: Returns elements from a sequence as long as a specified condition is true.
    - SkipWhile: Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements.

- **Ordering Operators:**
    - Order: Sorts the elements of a sequence in ascending order according to a key.
    - OrderByDescending: Sorts the elements of a sequence in descending order according to a key.
    - ThenBy: Performs a subsequent ordering of the elements in a sequence in ascending order.
    - ThenByDescending: Performs a subsequent ordering of the elements in a sequence in descending order.
    - Reverse: Inverts the order of the elements in a sequence.

- **Grouping Operators:**
    - GroupBy: Groups the elements of a sequence according to a specified key selector function.

- **Set Operators:**
    - Distinct: Removes duplicate elements from a sequence.
    - Union: Produces the set union of two sequences.
    - Intersect: Produces the set intersection of two sequences.
    - Except: Produces the set difference of two sequences.

- **Conversion Operators:**
    - AsEnumerable: Casts an IEnumerable to an IEnumerable<T>.
    - ToArray: Converts a sequence to an array.
    - ToList: Converts a sequence to a List<T>.
    - ToDictionary: Converts a sequence to a Dictionary<TKey, TValue> based on a key selector function.
    - OfType: Filters the elements of an IEnumerable based on a specified type.

- **Element Operators:**
    - First: Returns the first element of a sequence.
    - FirstOrDefault: Returns the first element of a sequence or a default value if no element is found.
    - Last: Returns the last element of a sequence.
    - LastOrDefault: Returns the last element of a sequence or a default value if no element is found.
    - Single: Returns the only element of a sequence and throws an exception if there is not exactly one element in the sequence.
    - SingleOrDefault: Returns the only element of a sequence or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence.
    - ElementAt: Returns the element at a specified index in a sequence.
    - ElementAtOrDefault: Returns the element at a specified index in a sequence or a default value if the index is out of range.

- **Quantifiers:**
    - Any: Determines whether any element of a sequence satisfies a condition.
    - All: Determines whether all elements of a sequence satisfy a condition.
    - Contains: Determines whether a sequence contains a specified element.

- **Aggregate Operators:**
    - Count: Counts the elements in a sequence.
    - LongCount: Counts the elements in a sequence, returning the count as a long.
    - Sum: Computes the sum of a sequence of numeric values.
    - Min: Returns the minimum value in a sequence.
    - Max: Returns the maximum value in a sequence.
    - Average: Computes the average of a sequence of numeric values.
    - Aggregate: Applies an accumulator function over a sequence.

- **Equality Operators:**
    - SequenceEqual: Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type.

- **Generation Operators:**
    - Empty: Returns an empty IEnumerable<T> with the specified type argument.
    - Repeat: Generates a sequence that contains one repeated value.
    - Range: Generates a sequence of integral numbers within a specified range.

Datatable Linq Operations

In C#, the DataTable class is part of the ADO.NET framework and is used to represent an in-memory table of data. LINQ (Language Integrated Query) can be used to perform various operations on a DataTable. LINQ provides a concise and expressive syntax for querying collections, including DataTables. Here are some common LINQ operations you can perform on a DataTable:

- **Filtering Rows:**
```
var filteredRows = from DataRow row in dataTable.Rows
                   where (int)row["Column1"] > 10
                   select row;
```

- **Sorting Rows:**
```
var sortedRows = from DataRow row in dataTable.Rows
                 orderby (string)row["ColumnName"] ascending
                 select row;
```

- **Selecting Specific Columns:**
```
var selectedColumns = from DataRow row in dataTable.Rows
                      select new
                      {
                          Column1 = (int)row["Column1"],
                          Column2 = (string)row["Column2"]
                      };
```

- **Grouping Rows:**
```
var groupedRows = from DataRow row in dataTable.Rows
                  group row by row["Column1"] into grouped
                  select new
                  {
                      Key = grouped.Key,
                      Count = grouped.Count()
                  };
```

- **Aggregating Data:**
```
var sum = dataTable.AsEnumerable().Sum(row => (int)row["Column1"]);
var average = dataTable.AsEnumerable().Average(row => (double)row["Column2"]);
```

- **Joining Tables:**
```
var result = from row1 in table1.AsEnumerable()
             join row2 in table2.AsEnumerable()
             on row1["CommonColumn"] equals row2["CommonColumn"]
             select new
             {
                 Column1 = row1["Column1"],
                 Column2 = row2["Column2"]
             };
```

Remember to include the necessary using directives at the beginning of your C# file:

```
using System.Data;
using System.Linq;
```

These examples assume that you have a DataTable named dataTable with appropriate columns. Adjust column names and types according to your specific scenario. Additionally, you can use the extension methods provided by the System.Data.DataSetExtensions assembly to enable LINQ on DataTable.

Security and Cryptography in c#

Cryptography is the science of creating messages that hide their meaning. In C# Cryptography can be used to protect sensitive information from unauthorized access or theft.

Here are some types of cryptography:

- **Secret Key Cryptography:** Also known as symmetric cryptography, this uses the same key for both encryption and decryption. Some examples of secret-key cryptographic algorithms include AES, 3DES, and RC4.
- **Public Key Cryptography:** Also known as asymmetric cryptography, this uses mathematical functions to create codes that are difficult to crack. Asymmetric encryption uses a public key for encryption and a private key for decryption. The public and private keys are mathematically related, so only the private key can decrypt data encrypted by the related public key.
- **Hash Functions:** This algorithm doesn't use a key.

In C# cryptography can be performed using the System.Security.Cryptography namespace. This namespace provides classes for encryption and decryption algorithms such as AES, RSA, and DES.

Symmetric encryption is mostly used for data confidentiality, while asymmetric encryption is mostly used for authentication and key exchange. Asymmetric encryption is slower than symmetric encryption, so it's not preferred for bulk data encryption.

Security and cryptography play crucial roles in software development, especially when dealing with sensitive data. In C#, you can leverage various libraries and classes to implement security and cryptographic features. Here's an overview of key aspects:

1. **Secure Coding Practices:**

- Always validate and sanitize user inputs to prevent common vulnerabilities like SQL injection and cross-site scripting (XSS).
- Use parameterized queries or stored procedures to interact with databases to prevent SQL injection attacks.
- Employ proper error handling and logging to identify and respond to potential security issues.

2. **Authentication and Authorization:**

- Utilize the ASP.NET Identity framework for user authentication and authorization.
- Implement role-based access control (RBAC) to restrict access to certain features or data based on user roles.

3. **Cryptography:**

- C# provides a System.Security.Cryptography namespace that includes various classes for cryptographic operations.
- Hashing: Use hash functions like SHA-256 for password storage. The SHA256 class can be used for this purpose.

```
using System.Security.Cryptography;
```

```
// ...
byte[] hashedBytes =
SHA256.Create().ComputeHash(Encoding.UTF8.GetBytes(inputString));
```

- Encryption/Decryption: For secure data transmission or storage, use symmetric (AES) or asymmetric (RSA) encryption.

```
using System.Security.Cryptography;
// ...

// Example using AES encryption
using (Aes aesAlg = Aes.Create())
{
   // Key and IV should be securely generated and stored
   aesAlg.Key = keyBytes;
   aesAlg.IV = ivBytes;

   // Create an encryptor to perform the stream transform.
   ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);

   // Use a CryptoStream to perform encryption
     using (CryptoStream csEncrypt = new CryptoStream(encryptStream, encryptor,
CryptoStreamMode.Write))
   {
      // Write the data to be encrypted to the stream.
      csEncrypt.Write(dataBytes, 0, dataBytes.Length);
      csEncrypt.FlushFinalBlock();
   }
}
```

4. **SSL/TLS:**
- Use HTTPS (SSL/TLS) to encrypt data in transit. ASP.NET supports SSL/TLS for secure communication between clients and servers.

5. **Secure Password Storage:**
- Store passwords securely using hash functions with a unique salt for each user.

6. **Secure File Handling:**
- When dealing with files, validate file types, restrict file upload sizes, and use secure file handling practices.

7. **Secure Configuration Management:**
- Avoid hardcoding sensitive information like API keys or database connection strings. Use configuration files or secure key management systems.

8. **Logging and Auditing:**
- Implement logging and auditing to monitor and detect security events.

System.Security.Cryptography Class

In C#, the System.Security.Cryptography namespace provides classes for various cryptographic operations. Here are some key classes within this namespace:

1. **Hashing:**

- MD5: Represents the MD5 hash algorithm.
- SHA1: Represents the SHA-1 hash algorithm.
- SHA256, SHA384, SHA512: Represent the SHA-2 family hash algorithms.

Example of using SHA-256 for hashing:

```
using System.Security.Cryptography;

string inputString = "Hello, World!";
byte[] inputBytes = Encoding.UTF8.GetBytes(inputString);

using (SHA256 sha256 = SHA256.Create())
{
   byte[] hashedBytes = sha256.ComputeHash(inputBytes);
}
```

2. **Symmetric Encryption:**

- Aes: Represents the AES (Advanced Encryption Standard) algorithm for symmetric encryption.

Example of using AES for encryption:

```
using System.Security.Cryptography;

byte[] key = GenerateKey(); // Ensure a secure key generation method
byte[] iv = GenerateIV();   // Ensure a secure IV generation method

using (Aes aesAlg = Aes.Create())
{
   aesAlg.Key = key;
   aesAlg.IV = iv;

   // Create an encryptor to perform the stream transform.
   ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);

   // Use a CryptoStream to perform encryption
     using (CryptoStream csEncrypt = new CryptoStream(encryptStream, encryptor,
CryptoStreamMode.Write))
   {
      csEncrypt.Write(dataBytes, 0, dataBytes.Length);
```

```
        csEncrypt.FlushFinalBlock();
    }
}
```

3. **Asymmetric Encryption:**

- RSA: Represents the RSA algorithm for asymmetric encryption.

  Example of using RSA for encryption:

```
using System.Security.Cryptography;

byte[] dataToEncrypt = Encoding.UTF8.GetBytes("Hello, World!");

using (RSA rsa = RSA.Create())
{
            byte[] encryptedData = rsa.Encrypt(dataToEncrypt,
RSAEncryptionPadding.OaepSHA256);
}
```

4. **Digital Signatures:**

- RSA can also be used for creating and verifying digital signatures.

  Example of creating a digital signature:

```
using System.Security.Cryptography;

byte[] dataToSign = Encoding.UTF8.GetBytes("Hello, World!");

using (RSA rsa = RSA.Create())
{
    byte[] signature = rsa.SignData(dataToSign, HashAlgorithmName.SHA256,
RSASignaturePadding.Pkcs1);
}
```

  Example of verifying a digital signature:

```
using System.Security.Cryptography;

byte[] dataToVerify = Encoding.UTF8.GetBytes("Hello, World!");

using (RSA rsa = RSA.Create())
{
    bool verified = rsa.VerifyData(dataToVerify, signature,
HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
}
```

What is AES Encryption?

The AES Encryption algorithm is a Symmetric-key Block Cipher that uses a fixed-length block of 128 bits and key sizes of 128, 192, or 256 bits. It was developed by two Belgian cryptographers, Joan Daemen, and Vincent Rijmen, and was adopted as a U.S. Federal Information Processing Standard (FIPS) in 2001. AES Encryption is widely used in a variety of applications, including secure communication protocols, file encryption, and password protection.

The AES Encryption algorithm uses a series of mathematical operations to encrypt and decrypt data. The encryption process involves several rounds of substitution, permutation, and linear transformation operations. The key used for encrypting and decrypting the code is the same, which is why AES is known as a Symmetric-Key Algorithm. This means that the same key is used to encrypt and decrypt data.

Initialization Vector :- An initialization vector (IV) is used in the Advanced Encryption Standard (AES) to ensure that the same value encrypted multiple times will not always result in the same encrypted value. This is an added security layer.

Dynamic Type

In C# 4.0, a new type is introduced that is known as a dynamic type. It is used to avoid the compile-time type checking. The compiler does not check the type of the dynamic type variable at compile time, instead of this, the compiler gets the type at the run time. The dynamic type variable is created using the dynamic keyword.

Example:

```
dynamic value = 123;
```

Important Points:

- In most of the cases, the dynamic type behaves like object types.
- You can get the actual type of the dynamic variable at runtime by using GetType() method. The dynamic type changes its type at the run time based on the value present on the right-hand side. As shown in the below example.

Example:

```
// C# program to illustrate how to get the
// actual type of the dynamic type variable
using System;

class Demo {

   // Main Method
   static public void Main()
   {

      // Dynamic variables
      dynamic value1 = "Hello World";
      dynamic value2 = 123234;
      dynamic value3 = 2132.55;
      dynamic value4 = false;

      // Get the actual type of
      // dynamic variables
      // Using GetType() method
      Console.WriteLine("Get the actual type of value1: {0}",
                        value1.GetType().ToString());

      Console.WriteLine("Get the actual type of value2: {0}",
                        value2.GetType().ToString());

      Console.WriteLine("Get the actual type of value3: {0}",
                        value3.GetType().ToString());

      Console.WriteLine("Get the actual type of value4: {0}",
                        value4.GetType().ToString());
```

```
    }
}
```

Output:

```
Get the actual type of value1: System.String
Get the actual type of value2: System.Int32
Get the actual type of value3: System.Double
Get the actual type of value4: System.Boolean
```

When you assign a class object to the dynamic type, then the compiler does not check for the right method and property name of the dynamic type which holds the custom class object.

You can also pass a dynamic type parameter in the method so that the method can accept any type of parameter at run time. As shown in the below example.

Example:

```
// C# program to illustrate how to pass
// dynamic type parameters in the method
using System;

class Demo {

    // Method which contains dynamic parameters
    public static void addstr(dynamic s1, dynamic s2)
    {
        Console.WriteLine(s1 + s2);
    }

    // Main method
    static public void Main()
    {

        // Calling addstr method
        addstr("G", "G");
        addstr("Cat", "Dog");
        addstr("Hello", 1232);
        addstr(12, 30);
    }
}
```

Output:

```
GG
CatDog
Hello1232
42
```

- The compiler will throw an exception at runtime if the methods and the properties are not compatible.
- It does not support the intellisense in visual studio.
- The compiler does not throw an error on dynamic type at compile time if there is no type checking for dynamic type.

Here are some examples of when to use dynamic types in C#:

- Communicating with other dynamic languages
- Simplifying responses from API calls
- Creating libraries that can be used between languages
- Making generic solutions when speed isn't the main concern
- Coding using reflection or dynamic languages
- Working with COM objects
- Getting results out of the LinQ queries
- Handling a variety of user input or other types of data