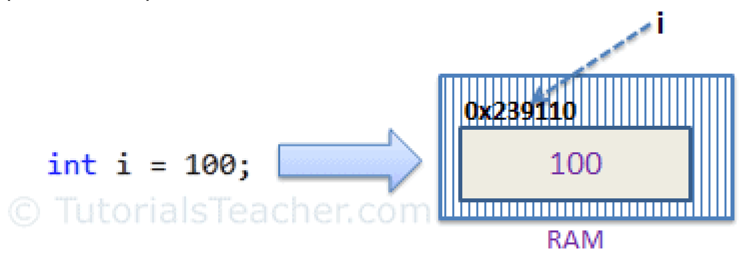
.NET & C#

.NET Framework
The .NET Framework is a software development platform developed by Microsoft.
it provides consistent programming environment for building and running various types of applications, including desktop applications, web applications, console applications, mobile applications.
It includes a runtime environment called the Common Language Runtime (CLR) and a rich set of class libraries. It supports multiple programming languages such as C#, VB.NET, and F#, and offers features like memory management, security, and exception handling.

Common Language Runtime (CLR)
The Common Language Runtime (CLR) is the execution environment provided by the .NET Framework. It manages the execution of .NET applications, providing services like memory management, code verification, security, garbage collection, and exception handling.
One of the key features of the CLR is the Just-In-Time (JIT) compiler. When a .NET application is executed, the CLR uses the JIT compiler to convert the Intermediate Language (IL) code a low-level platform-agnostic programming language into native machine code specific to the system the application is running on. This process happens at runtime, hence the term "Just-In-Time". This allows .NET applications to be platform-independent until they are executed, providing a significant advantage in terms of portability and performance.

Value Type and Reference Type
In .NET, data types are divided into two categories: value types and reference types. The primary difference between them lies in how they store their data and how they are handled in memory.
Value types directly contain their data and are stored on the stack. They include primitive types such • int •bool • byte • char • decimal • double • enum • float • long • sbyte • short • struct • uint • ulong • ushort , • enum, and • struct. When a value type is assigned to a new variable, a copy of the value is made. Therefore, changes made to one variable do not affect the other. All the value types derive from System.ValueType, which in-turn, derives from System.Object.
The system stores 100 in the memory space allocated for the variable i. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':

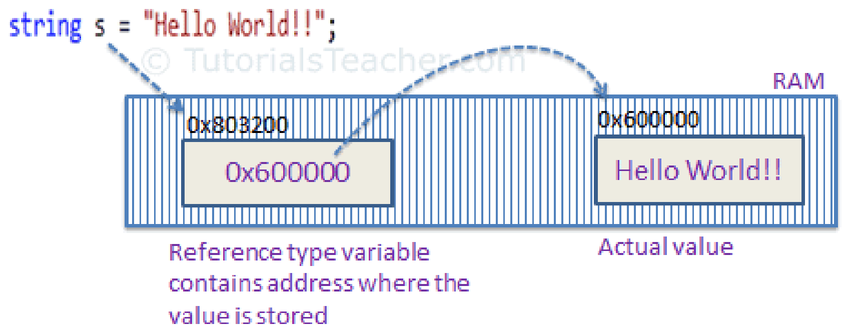


Reference types, on the other hand Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored on the heap. They include types such as class, interface, delegate, string, and array. When a reference type is assigned to a new variable, the reference is copied, not the actual data. Therefore, changes made to one variable will affect the other, as they both point to the same data.

For example, consider the following string variable:

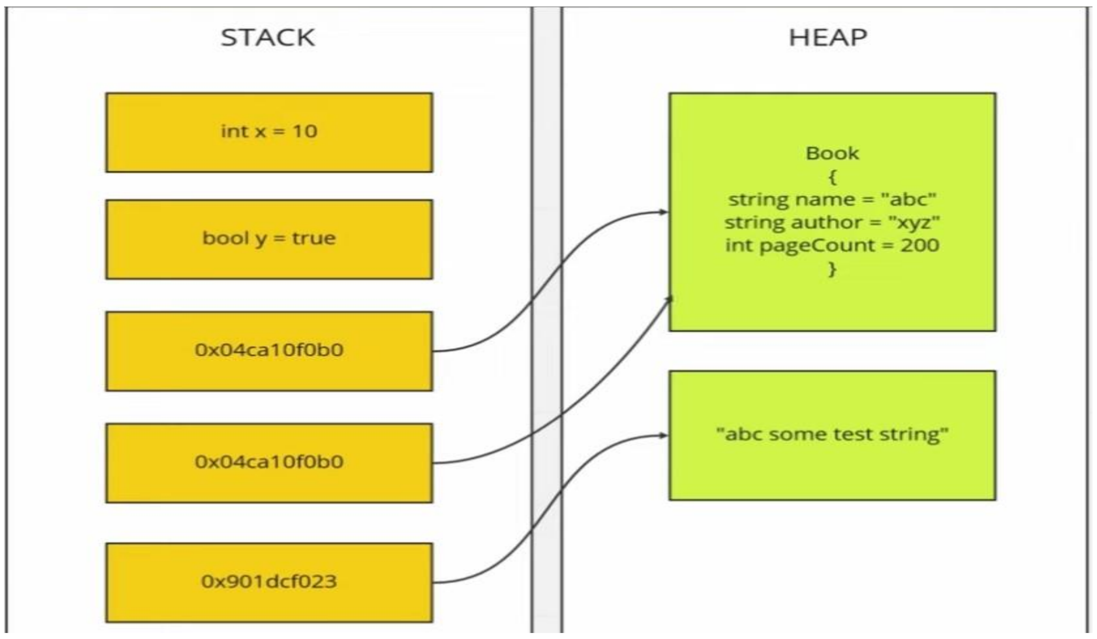
string s = "Hello World!!";

The following image shows how the system allocates the memory for the above string variable.


string s = "Hello World!!";
© TutorialsTeacher.com
RAM
0x803200
0x600000
0x600000
Hello World!!
Reference type variable contains address where the value is stored
Actual value

Memory Allocation of Reference Type Variable

As you can see in the above image, the system selects a random location in memory (0x803200) for the variable s. The value of a variable s is 0x600000, which is the memory address of the actual data value. Thus, reference type stores the address of the location where the actual value is stored instead of the value itself.



Passing Reference Type Variables

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the variable's address. So, If we change the value of a variable in a method, it will also be reflected in the calling method.

Example:

```
static void ChangeReferenceType(Student std2)
{
   std2.StudentName = "Steve";
}
static void Main(string[] args)
{
   Student std1 = new Student();
   std1.StudentName = "Bill";
   ChangeReferenceType(std1);
   Console.WriteLine(std1.StudentName);
}
```

Output:

```
Steve
```

In the above example, we pass the Student object std1 to the ChangeReferenceType() method. Here, it actually pass the memory address of std1. Thus, when the ChangeReferenceType() method changes StudentName, it is actually changing StudentName of std1 object, because std1 and std2 are both pointing to the same address in memory.

String is a reference type, but it is immutable. It means once we assigned a value, it cannot be changed. If we change a string value, then the compiler creates a new string object in the memory and point a variable to the new memory location. So, passing a string value to a function will create a new variable in the memory, and any change in the value in the function will not be reflected in the original value, as shown below.

Example: Passing String

```
static void ChangeReferenceType(string name)
{
   name = "Steve";
}
```

```
static void Main(string[] args)
{
   string name = "Bill";
   ChangeReferenceType(name);
   Console.WriteLine(name);
}
Output:
Bill
```

ref out in

In C#, we have various techniques available for controlling the passing of arguments to methods. Among these, the keywords ref, out, and in play a particularly important role. These parameter modifiers allow you to specify the way data is transferred between methods, although they have some limitations.

Ref

The ref keyword is used to pass an argument to a method by reference, rather than by value. This means that if a method changes the value of a ref parameter, this change will also be visible to the variable passed by the caller.
It's important to highlight that to use a ref parameter, both the method definition and the calling method must explicitly use the ref keyword. Also, the variable passed as
a ref parameter must be initialized before being passed to the method

```
public void SampleMethod(ref int x)
{
   x = 10;
}
int num = 1;
SampleMethod(ref num);
Console.WriteLine(num); // Prints 10
```

Out

The out keyword is used to pass arguments by reference. Unlike ref, however, out does not require that the input variable be initialized before being passed. Also, the method must necessarily assign a value to the out parameter before it concludes its execution.

```
public void SampleMethod(out int x)
{
   x = 10;}
```

```
int num;
SampleMethod(out num);
Console.WriteLine(num); // Prints 10
```

In

The in keyword, introduced in C# 7.2, is used to pass an argument to a method by reference but prevents the value from being modified within the method. This is particularly useful when working with large structures, because it avoids the overhead of copying, while at the same time ensuring the integrity of the data.

```
public void SampleMethod(in int x)
{
  // x = 10; // This will generate an error
  Console.WriteLine(x);
}

int num = 1;
SampleMethod(in num); // Prints 1
```

Limitations

Despite their usefulness, the keywords ref, out, and in have some limitations:

- They cannot be used in async methods defined with the async modifier.
- They cannot be used in iterator methods that include a yield return or yield break statement.
- The first argument of an extension method cannot have the in modifier unless that argument is a struct.
- They cannot be used in the first argument of an extension method in which that argument is a generic type, even when that type is constrained to be a struct.

The restrictions also extend to extension methods:

- The out keyword cannot be used in the first argument of an extension method.
- The ref keyword cannot be used in the first argument of an extension method, unless the argument is a struct or an unconstrained generic type.
- The in keyword cannot be used unless the first argument is a struct. Furthermore, it cannot be used in any generic type, even when it is constrained to be a struct.

Additionally, it's important to remember that members of a class cannot have signatures that differ only by ref, in, or out. In practice, a compiler error will occur if the only difference between two members of a type is that one of them has a ref parameter and the other has an out or in parameter.

⛔ **Compiler Error CS0663**

Cannot define overloaded methods that differ only on ref and out.

```
public class TestClass
{
  public void SampleMethod(out int i) { }
  public void SampleMethod(ref int i) { }
}
```

Overloading is legal, however, if one method takes a ref, in, or out argument and the other has none of those modifiers, like this:

```
public class TestClass1
{
   public void SampleMethod(int i) { }
   public void SampleMethod(ref int i) { }
}
public class TestClass2
{
   public void SampleMethod(int i) { }
   public void SampleMethod(out int i) => i = 5;
}
public class TestClass3
{
   public void SampleMethod(int i) { }
   public void SampleMethod(in int i) { }
}
```

Conclusion

The ref, out, and in keywords provide detailed control over how arguments are passed to methods. Understanding and using them correctly can enhance code efficiency and safety. However, it's crucial to be aware of their limitations and use them cautiously, as improper use can render the code complex and hard to manage.

Attributes in .NET

Attributes in .NET are powerful constructs that allow developers to add metadata—additional descriptive information—to various elements in the code, such as classes, methods, properties, and more. This metadata can be accessed at runtime using reflection, allowing for dynamic and flexible programming.

Attributes have square brackets [] and are placed above the code elements they're related to. They can be utilised to control behaviour, provide additional information, or introduce extra functionality.

Assemblies
Assemblies are the building blocks of .NET applications. They are self-contained units that contain compiled code (executable or library), metadata, and resources.
An exe (executable) file contains an application's entry point and is intended to be executed directly. It represents a standalone program.
On the other hand, a dll (dynamic-link library) file contains reusable code that can be referenced and used by multiple applications. It allows for

The System.Reflection namespace provides classes and methods to inspect and manipulate metadata, types, and assemblies at runtime. It enables developers to dynamically load assemblies, create instances, invoke methods, and perform other reflection-related operations.

The Common Type System (CTS) in .NET is a standard that defines how types are declared, used, and managed within the .NET environment. It ensures interoperability between different programming languages that target the .NET Framework or .NET

Garbage collection (GC) in .NET is an automatic memory management system that handles the allocation and deallocation of memory for objects in .NET. Its primary purpose is to prevent memory leaks and simplify memory management for developers.

Delegates in .NET are type-safe function pointer in C#. It allows you to encapsulate a method with a specific signature and return type, enabling you to pass methods as parameters or store them in variables. Delegates provide the foundation for events and callback methods, making your code more modular and flexible.
The signature of the delegate must match the signature of the function, the delegate points to, otherwise you get a compiler error. This is the reason delegates are called as type safe function pointers.
A Delegate is similar to a class. You can create an instance of it, and when you do so, you pass in the function name as a parameter to the delegate constructor, and it is to this function the delegate will point to.

```
// Declare a delegate type that can point to methods with no parameters and no return type
public delegate void Notify();

class Program
{
// A method that matches the delegate signature.
public static void SendMessage()
{
Console.WriteLine("Message sent!");
}
static void Main()
{
// Create an instance of the delegate pointing to SendMessage.
Notify notifyDelegate = SendMessage;
// Invoke the delegate.
notifyDelegate();
}
}
```

Key Properties of Delegates:

Type Safety: Delegates are type-safe, meaning that they must match the signature (return type and parameters) of the method they point to.

Method Encapsulation: Delegates encapsulate methods, allowing you to treat methods as objects.

Multicast: Delegates can be combined, so they can invoke multiple methods in a single call.

```
public delegate void LogHandler(string message);

class Logger
{
    // Multicast delegate example.
    public static void LogToConsole(string message)
    {
        Console.WriteLine($"Console: {message}");
    }
    public static void LogToFile(string message)
    {
        // Simulate logging to a file.
        Console.WriteLine($"File: {message}");
    }
    static void Main()
    {
        // Multicast delegate pointing to two methods.
        LogHandler log = LogToConsole;
        log += LogToFile;
        // Invoke the delegate (both methods are called).
        log("Multicast delegate example.");
    }
}
```

Flexibility: Delegates can be passed as parameters to methods, stored in variables, and returned from methods.

Event is a mechanism that enables an object (the publisher) to notify other objects (the subscribers) when something of interest occurs. Events are built upon delegates and they add an additional layer of abstraction and encapsulation. Specifically, an event restricts direct access to the underlying delegate, preventing external code from resetting or invoking the delegate directly.

```
public delegate void NotifyEventHandler();

class Publisher
{
    // Declare an event based on the delegate.
    public event NotifyEventHandler OnNotify;
    // Method to raise the event.
    public void RaiseEvent()
    {
        if (OnNotify != null)  // Check if there are subscribers.
        {
            Console.WriteLine("Raising event...");
            OnNotify();  // Invoke the delegate, notifying subscribers.
        }
    }
}
class Subscriber
{
    public void HandleNotification()
    {
        Console.WriteLine("Subscriber received the notification.");
    }
}
class Program
{
    static void Main()
    {
        Publisher publisher = new Publisher();
        Subscriber subscriber = new Subscriber();
        // Subscribe to the event using +=
        publisher.OnNotify += subscriber.HandleNotification;
        // Trigger the event, notifying all subscribers.
        publisher.RaiseEvent();
        // Unsubscribe from the event using -=
        publisher.OnNotify -= subscriber.HandleNotification;
    }
}
```

Key Properties of Events:
Encapsulation: Events expose only the functionality to subscribe (+=) and unsubscribe (=) listeners, restricting any direct invocation or resetting of the delegate.
Notification Mechanism: Events allow one object (the publisher) to notify other objects (subscribers) when something of interest happens.
Class/Struct Limitation: Events can only be declared within classes or structs, ensuring that their use is tied to object-oriented programming principles.

Anonymous method is a method without a name. Introduced in C# 2, they provide us a way of creating delegate instances without having to write a separate method

```
public delegate void Print(int value);

static void Main(string[] args)
```

```
{
    Print print = delegate(int val) {
        Console.WriteLine("Inside Anonymous method. Value: {0}", val);
    };

    print(100);
}
```

Output:

```
Inside Anonymous method. Value: 100
```

Lambda expressions in C# are a concise way to define anonymous functions. They are particularly useful for writing short, inline code blocks, especially when working with LINQ queries, event handlers, or functional programming paradigms.
Lambda expressions can be used anywhere you would have used an anonymous method or a strongly typed delegate (typically with far fewer keystrokes).
Under the hood, the C# compiler translates the expression into a standard anonymous method making use of the Predicate delegate type (which can be verified using ildasm.exe or reflector.exe).
Specifically, the following code statement:

```
// This lambda expression...
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

is compiled into the following approximate C# code:

```
// ...becomes this anonymous method.
List<int> evenNumbers = list.FindAll(delegate (int i)
{
return (i % 2) == 0;
});
```

Thus, assuming you have defined the following delegate type: public delegate string VerySimpleDelegate();
VerySimpleDelegate d = new VerySimpleDelegate(() => {return "Enjoy your string!";});

Using the new expression syntax, the previous line can be written like this:

VerySimpleDelegate d2 = new VerySimpleDelegate(() => "Enjoy your string!");

which can also be shortened to VerySimpleDelegate d3 = () => "Enjoy your string!";

Be aware, however, this new shortened syntax can be used anywhere at all, even when your code has nothing to do with delegates or events. So, for example, if you were to build a trivial class to add two numbers, you might write the following:

```
class SimpleMath
{
```

```
public int Add(int x, int y)
{
return x + y;
}
}
```

Alternatively, you could now write code like the following:

```
public int Add(int x, int y) => x + y;
```

Func, Action, and Predicate

In C#, Func, Action, and Predicate are all built-in generic delegates used to represent methods as parameters or to define method signatures. The key difference lies in their return type and the number of parameters they can accept.

Action Delegate:

- Represents a method that takes zero or more input parameters but does not return any value (i.e., its return type is void).
- Syntax: Action<T1, T2, ...> where T1, T2, etc., are the types of the input parameters.
- Example:

Code

```
Action<string> greet = name => Console.WriteLine($"Hello, {name}!");
greet("World"); // Output: Hello, World!
```

Func Delegate:

- Represents a method that takes zero or more input parameters and returns a value of a specified type.
- Syntax: Func<T1, T2, ..., TResult> where T1, T2, etc., are the types of the input parameters, and TResult is the type of the returned value.
- Example:

Code

```
Func<int, int, int> add = (a, b) => a + b;
int sum = add(5, 3); // sum = 8
```

Predicate Delegate:

- A specialized type of Func delegate that represents a method taking exactly one input parameter and always returning a bool value. It's commonly used for evaluating conditions.
- Syntax: Predicate<T> where T is the type of the single input parameter.
- Example:

Code

```
Predicate<int> isEven = x => x % 2 == 0;
bool result = isEven(4); // result = true
```

- Use Action when you need to pass a method that performs an operation but doesn't return a result.

- Use Func when you need to pass a method that performs an operation and returns a result of any type.
- Use Predicate when you need to pass a method that takes a single argument and returns a boolean value, typically for filtering or validation purposes.

Extension methods in C# allow developers to add new methods to existing types without modifying their source code. They are defined as static methods within a static class, and the first parameter of the extension method specifies the type being extended, preceded by the 'this' keyword. Extension methods enable adding functionality to types without inheritance or modifying the type hierarchy, making it easier to extend third-party or framework classes.

The using keyword has two major uses:
The using statement in C# is used for the automatic disposal of unmanaged resources, such as database connections, file streams, or network sockets, that implement the IDisposable interface. It ensures that the Dispose method of the resource is called when the code block within the using statement is exited, even in the presence of exceptions. It simplifies resource management and helps prevent resource leaks by providing a convenient syntax for working with disposable objects.
The *using statement* can be used to reference a variable or the result from a method, and at the end of the scope defined by the *using statement*, the `Dispose` method gets invoked:

```
private static void TraditionalUsingStatement()
{
  using (var r = new AResource())
  {
    r.UseIt();
  } // r.Dipose is called
}
```

Behind the scenes, the compiler creates code using *try/finally* to make sure `Dispose`is also called when an exception is thrown:

```csharp

private static void TraditionalUsingStatementExpanded()
{
  var r = new AResource();
  try
  {
    r.UseIt();
  }
  finally
  {
    r.Dispose();
  }
}
```

C# 8 now adds another variant to the *using* keyword – one that is related with the ***using statement***, the **using declaration**.
With the new C# 8 *using declaration*, the code with the *using statement can be simplified. Curly brackets are no longer needed. At the end of the scope of the variable r (which is here the end of the method), the* `Dispose` *method is invoked. Here, the compiler also creates a try/finally* block to make sure `Dispose` is called if errors occur.

```csharp
private static void NewWithUsingDeclaration()
{
  using var r = new AResource();
  r.UseIt();
}
```

The using directive creates an alias for a namespace or imports types defined in other namespaces
Inversion of Control (IoC) is a design principle that promotes loose coupling and modularity by inverting the traditional flow of control in software systems. Instead of objects creating and managing their dependencies, IoC delegates the responsibility of creating and managing objects to a container or framework. In .NET, IoC is commonly achieved through frameworks like Dependency Injection (DI) containers, where dependencies are injected into objects by the container, enabling flexible configuration and easier testing.

String vs StringBuilder
A **string** is an immutable object. This means once a string object is created, its value cannot be changed. When you modify a string (for example, by concatenating it with another string), a new string object is created in memory to hold the new value. This can lead to inefficiency if you're performing a large number of string manipulations.
On the other hand, **StringBuilder** is mutable. When you modify a StringBuilder object, the changes are made to the existing object itself, without creating a new one. This makes StringBuilder more efficient for scenarios where you need to perform extensive manipulations on a string.

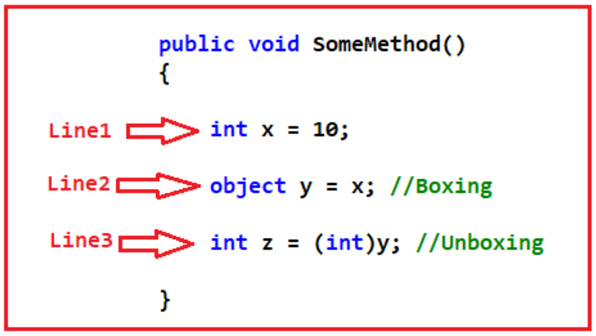
The .NET Standard is a formal specification that defines a common set of APIs that must be available on all .NET implementations. It serves as a bridge between different .NET platforms such as .NET Framework, .NET Core, and Xamarin.
By targeting the .NET Standard, developers can create libraries that can be used across multiple .NET platforms without the need for platform-specific code. The .NET Standard enables code sharing and simplifies the development process by providing a consistent set of APIs that are available across different platforms.

ASP.NET is a server-side web-application framework designed for web development to produce dynamic web pages. It was developed by Microsoft to allow programmers to build dynamic web sites, applications and services. The name stands for Active Server Pages
ASP stands for **A**ctive **S**erver **P**ages
ASP is a development framework for building web pages.
ASP supports many different development models:

- Classic ASP
- ASP.NET Web Forms
- ASP.NET MVC
- ASP.NET Web Pages
- ASP.NET API
- ASP.NET Core

Middleware in ASP.NET Core is a component in the application pipeline used to handle requests and responses.Each middleware component can inspect, modify, or pass the request to the next middleware in the pipeline. Middleware components are registered in a specific order, and the request flows through them in that order.

Boxing: Boxing is the process of converting a value type (like int, double, struct) to a reference type (object).
Unboxing: Unboxing is the reverse process of boxing, It involves explicitly converting a reference type (object) into a value type.



```
public void SomeMethod()
{

    int x = 10;

    object y = x; //Boxing

    int z = (int)y; //Unboxing

}
```

In C# try, catch, and finally blocks are used for exception handling. While the try block is mandatory, the catch and finally blocks are optional and can be skipped depending on the desired behavior.

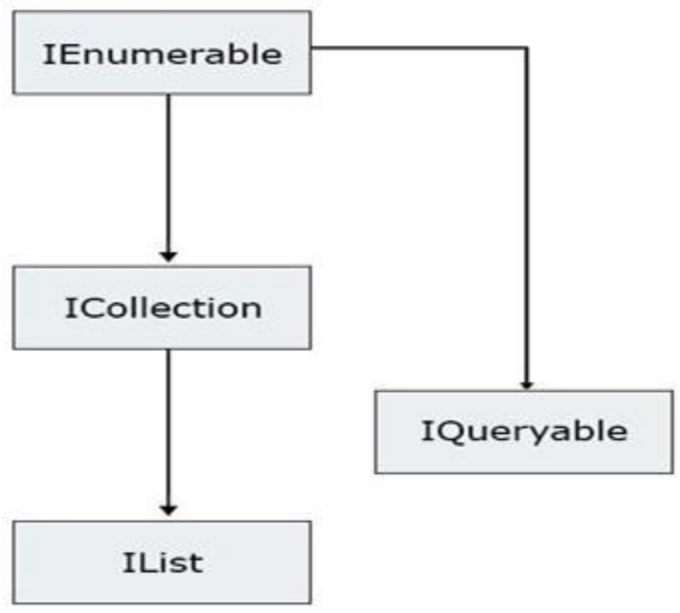
1. The try block:
    - This block contains the code that might potentially throw an exception. It is the only mandatory part of the try-catch-finally construct.
2. The catch block(s):
    - catch blocks are used to handle specific types of exceptions that might be thrown within the try block.
    - Skipping catch: A try block can exist without a catch block if the intention is to let any exceptions thrown within the try block propagate up the call stack to be handled by a higher-level exception handler. This is often done when the current method cannot meaningfully recover from the exception, or when a finally block is needed for resource cleanup regardless of whether an exception is caught.
3. The finally block:
    - The finally block contains code that will always execute, regardless of whether an exception was thrown in the try block or caught by a catch block.
    - Skipping finally: A finally block can be skipped if there are no resources to clean up or actions that must be performed regardless of exception occurrence. For example, if all resource management is handled by using statements (which implicitly provide finally-like behavior for IDisposable objects), a separate finally block might not be necessary.

In summary:

- You cannot skip the try block.
- You can skip the catch block(s) if you want exceptions to propagate or if you are only concerned with cleanup actions in a finally block.
- You can skip the finally block if there are no cleanup actions or other code that needs to execute unconditionally.

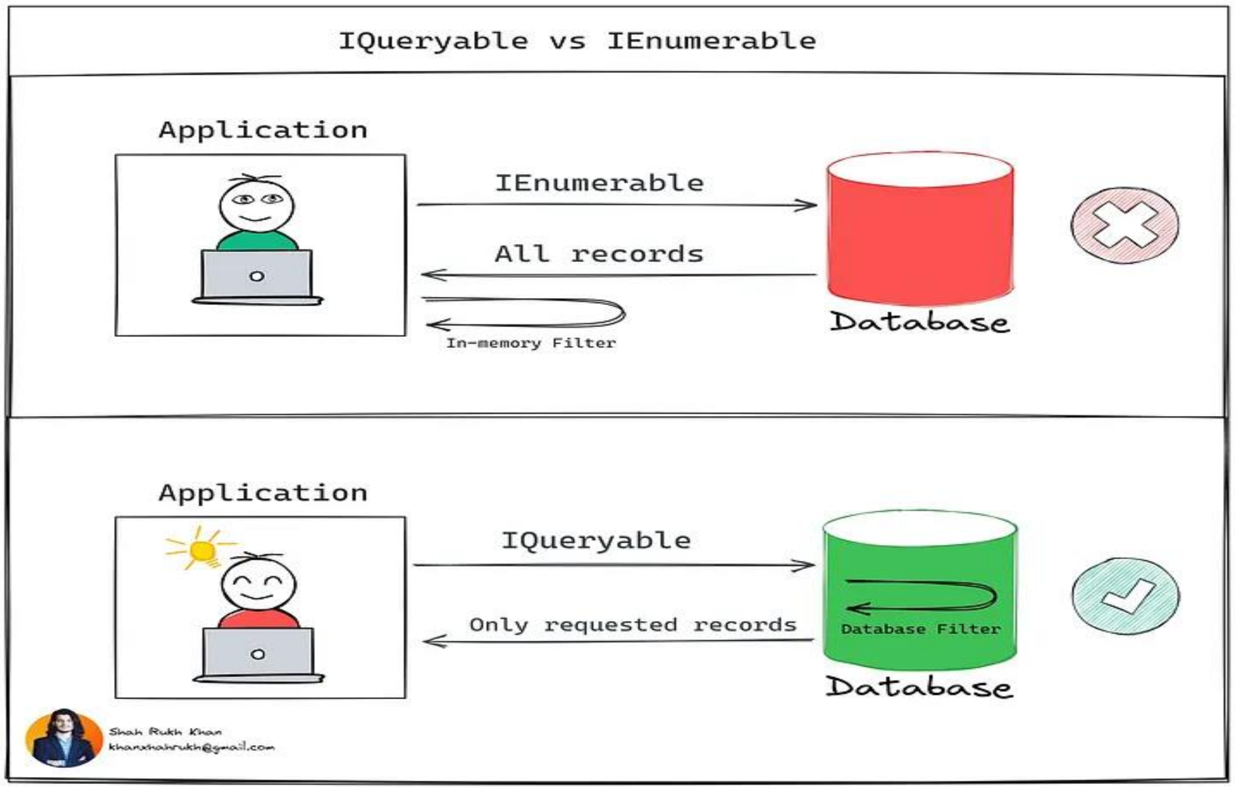
IEnumerable is a list or a container which can hold some items. You can iterate through each element in the IEnumerable. You can not edit the items like adding, deleting, updating, etc. instead you just use a container to contain a list of items. It is the most basic type of list container. All you get in an IEnumerable is an enumerator that helps in iterating

over the elements. An IEnumerable does not hold even the count of the items in the list, instead, you have to iterate over the elements to get the count of items.
ICollection is another type of collection, which derives from IEnumerable and extends it's functionality to add, remove, update element in the list. ICollection also holds the count of elements in it and we does not need to iterate over all elements to get total number of elements.
IList extends ICollection. An IList can perform all operations combined from IEnumerable and ICollection, and some more operations like inserting or removing an element in the middle of a list.



IEnumerable and IQueryable are both interfaces in .NET used for working with collections of data, but they differ significantly in their execution and intended use cases.
IEnumerable is designed for working with data that is already in memory. When you apply LINQ queries (like Where, OrderBy, etc.) to an IEnumerable, the entire collection is loaded into memory first, and then the filtering and sorting operations are performed on that in-memory data.

IQueryable is specifically designed for querying data from external sources, such as databases or web services. It translates LINQ queries into the native query language of the data source (e.g., SQL for a database)



Deferred vs immediate execution
In LINQ (Language Integrated Query), deferred execution means that the execution of a query is postponed until its results are actually needed. This contrasts with immediate execution, where the query runs as soon as it is defined.
Key aspects of LINQ deferred execution:
Query Definition vs. Execution:
When you define a LINQ query using methods like Where, Select, or OrderBy, you are essentially building a query definition, not executing it immediately. The query is represented by an object that encapsulates the instructions for retrieving and transforming data.

Triggering Execution:
The query is executed only when its results are enumerated or materialized. Common ways to trigger execution include:
Iteration: Using a foreach loop to iterate over the query results.
Materialization Methods: Calling methods
like ToList(), ToArray(), ToDictionary(), Count(), First(), Single(), Any(), etc., which force the query to execute and return the results in a specific data structure or a single value.
Materialization refers to the process of turning a query or any IEnumerable sequence into a concrete data structure, such as a list or an array. .AsEnumerable() and .ToList() are two methods that materialize results but in different ways.

Benefits: Deferred execution can improve performance, especially with large datasets, by processing data on demand and potentially avoiding unnecessary computations or data retrieval. For example, if you only need the first element that satisfies a condition, the query might stop processing after finding that element.

Dispose and Finalize are methods used for releasing unmanaged resources. Finalize is called automatically by the garbage collector, while Dispose is called explicitly by the

developer. The Dispose method is part of the IDisposable interface, and it's the preferred method for releasing resources because it provides deterministic cleanup.
Deterministic cleanup refers to the process of ensuring that resources are released at a predictable and defined time, rather than relying on the unpredictable timing of garbage collection. This is crucial for managing resources like file handles, network connections, or database sessions, where immediate and reliable cleanup is necessary

Finalize:

- **Automatic Execution:**

The garbage collector calls Finalize before an object is destroyed, but the timing is non-deterministic.

- **Purpose:**

Primarily used for releasing unmanaged resources that the object holds.

- **Cost:**

Has a performance cost associated with it due to the garbage collector's involvement.

- **Implementation:**

Implemented as a destructor (using the syntax ~ClassName()) or by overriding the Finalize method.

Dispose:

- **Manual Execution:** Dispose is a method that developers must explicitly call to release resources.
- **Purpose:** Releases both managed and unmanaged resources.
- **Benefits:**
    - Provides deterministic resource cleanup, meaning resources are released when you explicitly call Dispose, unlike Finalize.
    - Faster than Finalize because it bypasses the garbage collector.
    - Supports the using statement for automatic resource management.
- **Implementation:** Implemented by classes that implement the IDisposable interface.

Key Differences:

- **Invocation:**

Finalize is called by the garbage collector, while Dispose is called by the developer.

- **Timing:**

Finalize is called non-deterministically, while Dispose is called deterministically when the developer chooses.

- **Scope:**

Finalize is mainly for unmanaged resources, while Dispose can release both managed and unmanaged resources.

- **Performance:**

Finalize has a performance overhead due to the garbage collector, while Dispose is generally faster.

Best Practices:

- Prefer using Dispose for resource management whenever possible, as it offers deterministic cleanup and better performance.
- Implement Finalize only when absolutely necessary, especially for releasing unmanaged resources that may not be released promptly by the garbage collector.
- When implementing Dispose, consider using the using statement for automatic resource management.
- Always call GC.SuppressFinalize(this) in your Dispose method to prevent the garbage collector from also finalizing the object, as this can lead to unnecessary overhead.

Operators:
?? Operator it is placed between two operands and returns the left operand only if its value is not null, otherwise it returns the right operand.

for & foreach
In C#, both for and foreach loops are used for iteration, but they differ in their primary use cases and how they manage iteration:
for Loop:
- **Explicit Control:**

The for loop provides explicit control over the initialization, condition, and iteration step. This allows for fine-grained control over the looping process.
- **Index-Based Access:**

It is commonly used when you need to access elements by their index or when you need to iterate a specific number of times.
- **Flexibility:**

for loops can be used with or without collections, making them suitable for various scenarios, including iterating a fixed number of times or performing actions based on an index.
- **Syntax:**

Code
```
  for (initialization; condition; iterator)
  {
    // code to execute
  }
```
foreach Loop:
- **Collection Iteration:**

The foreach loop is designed specifically for iterating over elements in collections that implement the IEnumerable interface (e.g., arrays, lists, dictionaries).
- **Simplicity and Readability:**

It simplifies iteration by automatically handling the traversal of elements, making the code more concise and readable, as you don't need to manage indices.
- **No Index Access:**

You cannot directly access elements by index within a foreach loop. If index-based operations are required, a for loop is more appropriate.

- **Immutability (of the iterated value):**

The foreach loop provides a read-only view of the elements during iteration; you cannot modify the iterated element itself within the loop.

- **Syntax:**

Code
```
  foreach (var item in collection)
  {
    // code to execute
  }
```
Choosing between for and foreach:

- Use foreach

when you need to iterate over all elements in a collection and do not require index-based access or modification of the collection during iteration. It prioritizes readability and simplicity.

- Use for

when you need explicit control over the iteration process, require index-based access to elements, or need to iterate a specific number of times regardless of a collection.
While performance differences can exist in specific scenarios (e.g., iterating over large arrays vs. List<T> with foreach), for most common use cases, the choice between for and foreach should be based on readability, intent, and whether index-based access or modification is required.

Traditional Get and Set Methods Properties or Automatic properties
In C#, the management of class data members can be achieved through traditional get and set methods, properties, or automatic properties, each offering different levels of control and conciseness.
1. Traditional Get and Set Methods:
This approach involves creating explicit public methods to access and modify private fields within a class.
Code
```
public class ExampleClass
{
  private string _name;

  public string GetName()
  {
    return _name;
  }

  public void SetName(string value)
  {
    _name = value;
    }
}
```
2. Properties:
Properties encapsulate the get and set accessors, providing a more convenient and idiomatic way to interact with private fields while still allowing for custom logic within the accessors.
Code
```
public class ExampleClass
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; } // Validation or other logic can be added here
    }
}
```
3. Automatic Properties:
Automatic properties, also known as auto-implemented properties, are a shorthand for properties when no additional logic is required within the get or set accessors. The compiler automatically generates the private backing field.
Code
```
public class ExampleClass
{
    public string Name { get; set; } // The private backing field is implicitly created
}
```
Comparison:

- **Control and Logic:**

Traditional methods and full properties allow for custom logic (validation, event raising, etc.) within their accessors. Automatic properties offer no such direct control, as they are purely for simple field access.

- **Conciseness:**

Automatic properties are the most concise, followed by full properties, and then traditional methods.

- **Encapsulation:**

All three approaches promote encapsulation by providing controlled access to internal data, preventing direct manipulation of private fields.

- **Flexibility:**

Properties (both full and automatic) offer greater flexibility than traditional methods in terms of features like different access modifiers for get/set, virtual/abstract properties, and init accessors for object construction.
Choosing the right approach:

- Use automatic properties when a simple getter and setter are needed without any custom logic.

- Use full properties when custom logic (validation, side effects) is required during getting or setting a value.
- **Traditional get and set methods**: are generally less preferred in modern C# development due to the cleaner syntax and features offered by properties, but they remain a valid option for specific scenarios.

Virtual

The **virtual** keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class.

Sealed

In C#, a sealed method is a method that cannot be overridden by any derived classes. This is achieved by using the `sealed` keyword in conjunction with the `override` keyword when defining the method in a derived class.

Here's how it works:

- **Virtual Method in Base Class:** A method in a base class must first be declared as virtual to allow derived classes to override it.

```
public class Animal
{
  public virtual void MakeSound()
  {
    Console.WriteLine("Generic animal sound");
  }
}
```

- **Sealed Override in Derived Class:** In a derived class, you can override this virtual method and then mark the overridden method as sealed. This prevents any further classes that inherit from this derived class from overriding this specific method.

```
public class Dog : Animal
{
  public sealed override void MakeSound()
  {
    Console.WriteLine("Bark!");
  }
}
```

- **Preventing Further Overrides:** If you then try to create another class that inherits from `Dog` and attempts to override `MakeSound`, it will result in a compilation error because the method was sealed in the `Dog` class.

```
// This will cause a compilation error
public class Puppy : Dog
{
  public override void MakeSound() // Error: Cannot override sealed member
  {
    Console.WriteLine("Whimper!");
    }
  }
```
Purpose of Sealed Methods:

- **Controlling Inheritance:**

Sealed methods provide fine-grained control over how specific behaviors are inherited and modified within a class hierarchy.

- **Preventing Unintended Modifications:**

They ensure that a particular implementation of a method remains consistent and cannot be altered by further derived classes, guaranteeing predictable behavior.

- **Optimization:**

While not the primary reason, sealing a method can sometimes allow the C# compiler to perform certain optimizations, as it knows the method's implementation will not change.

Sealed Class

A sealed class in C# is a class that cannot be inherited by other classes. This is achieved by using the sealed keyword in the class declaration.

Key characteristics of sealed classes:

- **Prevents Inheritance:**

  The primary purpose of sealing a class is to prevent any other class from deriving from it. This means you cannot create a subclass that inherits the members and behavior of a sealed class.

- **Ensures Immutability (of implementation):**

  By preventing inheritance, sealed classes ensure that their implementation cannot be altered or extended through derived classes. This is useful when you want to guarantee that a specific class's behavior remains consistent and cannot be modified by external code.

- **Can Improve Performance:**

  In some scenarios, sealing a class can provide minor performance benefits. The Common Language Runtime (CLR) can make certain optimizations when it knows a class cannot be inherited, as it doesn't need to account for potential overrides of virtual methods.

- **Used for Framework Design:**

  Sealed classes are often used in framework design to control the extensibility of certain components. This helps maintain the integrity and predictability of the framework.

Example:

Code

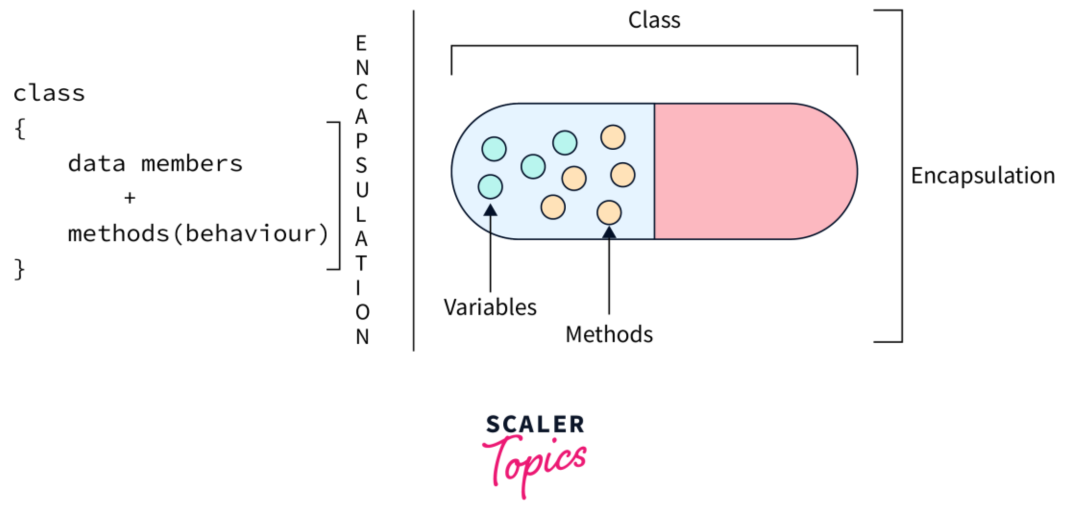
```
public sealed class MySealedClass
{
  public void DisplayMessage()
  {
    Console.WriteLine("This is a sealed class.");
  }
}

// Attempting to inherit from MySealedClass will result in a compile-time error:
// public class MyDerivedClass : MySealedClass // Error: 'MyDerivedClass': cannot inherit
from sealed class 'MySealedClass'
// {
// }
```

It is important to note that sealed cannot be applied to abstract classes or methods, as abstract implies the need for implementation in derived classes, which contradicts the purpose of sealed.

4 Pillars of object-oriented programming (OOP)
The 4 Pillars of object-oriented programming (OOP) are encapsulation, inheritance, polymorphism, and abstraction. These pillars are fundamental concepts that guide how objects interact and are structured in OOP languages

Encapsulation:
This principle involves wrapping data and the methods that operate on that data within a single unit, typically a class.Is like putting your things in a Capsule(class). It's a way to organize and protect your data and methods. You create a class, which is like a capsule, and you put your data (variables) and methods (functions) inside it. This keeps them together and organized. It acts as a protective shield, preventing direct external access to an object's internal state and ensuring that the data remains consistent and vali. This is often achieved through access modifiers like private and public properties.



- **public:**
  Allows unrestricted access to a class member from any part of the code.
- **private**
  Restricts access to a class member only within the same class, making it inaccessible from outside.
- **protected:**
  Limits access to a class member within the same class and its derived classes, enabling inheritance-related access.
- **internal:**
  Provides access to a class member within the same assembly (project) but restricts access from outside assemblies.

Inheritance:
This allows a new class (derived class or subclass) to inherit properties and behaviors from an existing class (base class or superclass). This promotes code reusability and establishes a hierarchical relationship between classes
Key aspects of C# inheritance:
Base Class and Derived Class:
The class whose members are inherited is known as the base class, while the class that inherits those members is called the derived class.
Syntax:
Inheritance is denoted using the colon (:) symbol followed by the base class name after the derived class name.

```
public class Vehicle
{
  public string brand = "Ford";
  public void Honk()
  {
    Console.WriteLine("Tuut, tuut!");
  }
}
public class Car : Vehicle // Car inherits from Vehicle
{
  public string modelName = "Mustang";
}
```

Single Inheritance:
C# supports single inheritance for classes, meaning a class can only inherit from one base class directly. However, inheritance is transitive, allowing for multi-level inheritance hierarchies where a base class can itself be a derived class of another.

Member Inheritance:
Derived classes implicitly gain all public and protected members of the base class. Private members of the base class are not directly accessible in the derived class. Constructors are not inherited.
Method Overriding:
Derived classes can modify the behavior of inherited methods using the override keyword, provided the base class method is marked with the virtual keyword.
base Keyword:
The base keyword is used within a derived class to access members (including constructors) of its immediate base class.
Achieving Multiple Inheritance:
While C# does not support multiple inheritance of classes directly, similar functionality can be achieved through interfaces. A class can implement multiple interfaces, allowing it to inherit behavior specifications from various sources.

Advantages:
Inheritance reduces code redundancy, promotes code reusability, and improves code organization and readability.
Disadvantages:
Deep inheritance hierarchies can lead to tight coupling, increased complexity, and the potential for the "fragile base class problem" where changes in the base class might inadvertently break derived classes.

Polymorphism:
Polymorphism allows treating objects of a derived class as objects of it's base class. When a derived class inherits from a base class, it includes all the members of the base class. All the behavior declared in the base class is part of the derived class. That enables objects of the derived class to be treated as objects of the base class.

There are two main types of polymorphism in C#:

- **Compile-time Polymorphism (Method Overloading):** This occurs when multiple methods within the same class share the same name but have different signatures (i.e., different numbers or types of parameters). The compiler determines which method to call based on the arguments provided during the method call.

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

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

- **Runtime Polymorphism (Method Overriding):** This is achieved through inheritance and virtual/override keywords. A base class declares a method as virtual, allowing derived classes to provide their own specific implementation of that method using the override keyword. When a method is called on an object whose runtime type is a derived class, the overridden method in the derived class is executed.

```
public class Animal
{
  public virtual void MakeSound()
  {
    Console.WriteLine("Generic animal sound.");
  }
  }

  public class Dog : Animal
  {
    public override void MakeSound()
    {
      Console.WriteLine("Woof!");
    }
  }

  public class Cat : Animal
  {
    public override void MakeSound()
    {
      Console.WriteLine("Meow!");
    }
  }
Animal myAnimal = new Dog(); // myAnimal is of type Animal, but holds a
Dog object
myAnimal.MakeSound(); // This will call the MakeSound method from the Dog
class at runtime.
```

Abstraction:
This focuses on hiding complex implementation details and exposing only the essential features of an object. It allows you to design and interact with objects based on what they do, rather than how they do it. Abstraction can be achieved using abstract classes and interfaces in C#.

In C#, abstraction is primarily achieved through:

- **Abstract Classes:**

Abstract classes are used when you need to define a common interface and some shared functionality for a group of related classes, while also requiring that certain behaviors be implemented specifically by each derived class. They offer a blend of interface-like contract enforcement and base class functionality sharing.
An abstract class is a restricted class that cannot be instantiated directly. It serves as a blueprint for other classes.

It can contain both abstract methods (methods without an implementation, marked with the abstract keyword) and concrete methods (methods with full implementation).

Derived classes that inherit from an abstract class must provide implementations for all abstract methods defined in the base abstract class.
Code

```
public abstract class Shape
{
  public abstract double GetArea(); // Abstract method - no implementation here
  public void DisplayMessage()
  {
    Console.WriteLine("This is a shape."); // Concrete method
  }
}

public class Circle : Shape
{
  private double radius;

  public Circle(double r)
  {
    radius = r;
  }

  public override double GetArea() // Implementation of the abstract method
  {
    return Math.PI * radius * radius;
  }
}
```

- **Interfaces:**
    - An interface defines a contract that classes can implement. It specifies a set of methods, properties, events, or indexers that a class must provide.
    - Interfaces do not contain any implementation details; they only define signatures.
    - A class can implement multiple interfaces, allowing for multiple inheritance of type.

Code

```
public interface IDisplayable
{
  void Display(); // Method signature
}

public class MyClass : IDisplayable
{
  public void Display() // Implementation of the interface method
  {
    Console.WriteLine("Displaying content from MyClass.");
    }
  }
```

Static Class
In C#, a static class is a special type of class that cannot be instantiated. This means you cannot create objects of a static class using the new keyword. Instead, all members (fields, properties, methods, events) within a static class must also be declared as static.

Here are the key characteristics of a static class in C#:

- **Cannot be instantiated:**
  You cannot create an instance of a static class.
- **Contains only static members:**
  All members (fields, properties, methods, etc.) within a static class must be declared with the static keyword.
- **Sealed:**
  Static classes are implicitly sealed, meaning they cannot be inherited by other classes.
- **Cannot contain instance constructors:**
  Since you cannot create instances, there is no need for instance constructors. However, they can have a static constructor to initialize static members.
- **Accessed directly via class name:**
  Static members of a static class are accessed directly using the class name, without creating an object.

When to use a static class:

Static classes are commonly used for:

- **Utility classes:**
  Housing helper methods or functions that perform operations not tied to a specific object instance (e.g., mathematical operations, string manipulation, validation).
- **Constants or immutable data:**
  Storing global constants or data that remains the same throughout the application's lifecycle.
- **Singleton pattern implementation:**
  While not strictly requiring a static class, the singleton pattern often utilizes static members to ensure only one instance of a class exists.

Example:
```
public static class MathHelper
{
    public static double Add(double a, double b)
    {
        return a + b;
    }

    public static double Multiply(double a, double b)
    {
        return a * b;
    }
}
// Usage:
double sum = MathHelper.Add(5.0, 3.0);
double product = MathHelper.Multiply(4.0, 2.0);
```

Static Constructor
In C#, a static constructor is a special type of constructor used to initialize static data or perform a specific action that needs to be executed only once for a class or struct

Key characteristics of static constructors:

- **Declaration:**

They are declared using the static keyword and do not have an access modifier (e.g., public, private).

- **Parameters:**

They cannot take any parameters.

- **Invocation:**

They are invoked automatically by the Common Language Runtime (CLR) at most once, before the first instance of the class is created or any static members of the class are accessed. The exact timing within this window is determined by the CLR.

- **Inheritance and Overloading:**

They cannot be inherited or overloaded. A class or struct can only have one static constructor.

- **Purpose:**

Their primary purpose is to initialize static fields, especially those that require complex initialization logic or depend on other static members. They are also useful for tasks like setting up logging, creating singleton instances, or performing runtime checks that cannot be done at compile time.

- **Limitations:**

They cannot access non-static members of the class. If a static constructor throws an exception, it is not invoked again, and the type remains uninitialized for the lifetime of the application domain.
Example:
Code
```
public class MyClass
{
    // Static field to be initialized by the static constructor
    private static int _staticCounter;

    // Static constructor
    static MyClass()
    {
        _staticCounter = 100; // Initialize static data
        Console.WriteLine("Static constructor called.");
    }

    // Instance constructor
    public MyClass()
    {
        Console.WriteLine("Instance constructor called.");
    }

    // Static method to access static data
    public static void DisplayStaticCounter()
    {
        Console.WriteLine($"Static Counter: {_staticCounter}");
    }
}
```

In this example, the static MyClass() constructor initializes _staticCounter to 100. This initialization happens automatically before the first MyClass object is created or DisplayStaticCounter() is called.

Namespace

In C#, a namespace is a mechanism used to organize and group related code elements, such as classes, interfaces, structs, enums, and delegates, into a logical hierarchy. It provides a way to define a scope for these elements, preventing naming conflicts and improving code readability and maintainability, especially in larger projects or when working with external libraries

Compile-time Constants (const)

Definition: These are values whose exact value is known at the time the code is compiled. The compiler directly substitutes the constant's value into the code wherever it is used.

Keyword: Declared using the const keyword.

Characteristics:

Must be initialized at the time of declaration.

Cannot be changed after compilation.

Can only be of primitive types (like int, string, bool, etc.) or null for reference types.

Are implicitly static.

Example**:**

```
const int MaxValue = 100;
const string Greeting = "Hello, World!";
```

Runtime Constants (readonly)

Definition: These are values that can be assigned at runtime (e.g., in the constructor of a class) but cannot be changed after that initial assignment. Their value is not known until the program executes.

Keyword: Declared using the readonly keyword.

Characteristics:

Can be initialized at declaration or in the constructor of the class or struct they belong to.

Cannot be changed after the initial assignment.

Can be of any data type, including custom classes or structs.

Example**:**

```
class MyClass
{
  public readonly int Id;
  public readonly DateTime CreationTime;

  public MyClass(int id)
  {
    Id = id;
    CreationTime = DateTime.Now; // Value determined at runtime
  }
}
```

Key Differences Summarized:

Value Determination:

const values are determined at compile time; readonly values are determined at runtime.
Initialization:
const must be initialized at declaration; readonly can be initialized at declaration or in a constructor.
Mutability:
Both are immutable after their initial assignment, but const is fixed at compilation, while readonly is fixed during instance creation or at declaration.
Applicability:
const is limited to primitive types and null; readonly can be used with any data type.
Static Implication:
const is implicitly static; readonly can be static or instance-level.

Class
C# supports various types of classes, each serving a specific purpose in object-oriented programming:

- **Concrete/Regular Class:**

This is the most common type of class, used to create objects that can be instantiated and contain both data (fields) and behavior (methods). Without specifying **public** the **class is** implicitly internal . This **means** that the **class is** only
visible inside the same assembly(project). When you specify **public** , the **class is** visible outside the assembly

- **Abstract Class:**

An abstract class serves as a blueprint for other classes and cannot be instantiated directly. It can contain both abstract members (which must be implemented by derived classes) and non-abstract members. Abstract classes are used to define a common interface and shared functionality for a hierarchy of related classes.
it is perfectly fine (and common) for abstract classes to define any number of constructors that are called *indirectly* when derived classes are allocated

- **Sealed Class:**

A sealed class cannot be inherited by other classes. The sealed keyword prevents further extension of the class, often used for security, performance optimization, or to ensure the integrity of a class's implementation.

- **Static Class:**

A static class can only contain static members (fields, methods, properties, events). It cannot be instantiated using the new keyword, and its members are accessed directly through the class name. Static classes are commonly used for utility classes or to group related helper functions that don't require an instance.

- **Partial Class:**

A partial class allows the definition of a single class to be split across multiple source files. This is useful for organizing large classes, enabling multiple developers to work on different parts of the same class simultaneously, or integrating code generated by tools with custom code.

- **Nested Class:**

A nested class is a class defined within another class. It is used to logically group related functionality and can access the members of its containing class. Nested classes can be declared with various access modifiers (e.g., public, private).

- **Generic Class:**

A generic class is a class that works with type parameters, allowing it to operate on different data types without requiring separate implementations for each type. This promotes code reusability and type safety.

Classes that can not be instantiated

Abstarct class can not be instantiated

A class that has only private constructors cannot be instantiated((other than that class itself))

When a class is declared as static, it is an indication that this class contains only static members (i.e. static fields, methods, properties) and cannot be instantiated

Array, ArrayList, and List<T>

In C#, Array, ArrayList, and List<T> are used to store collections of data, but they differ in their characteristics and use cases.

1. Array:

- **Fixed Size:** An array's size is determined at its creation and cannot be changed.
- **Homogeneous:** It stores elements of a single, specified data type.
- **Performance:** Offers the fastest access to elements due to contiguous memory allocation and direct indexing.
- **Declaration Example:**

Code

```
int[] numbers = new int[5]; // Declares an array of 5 integers
```

2. ArrayList:

- **Dynamic Size:** Can grow or shrink dynamically as elements are added or removed.
- **Heterogeneous:** Can store elements of different data types (as object types, requiring boxing/unboxing for value types).
- **Non-Generic:** Part of System.Collections namespace.
- **Performance:** Generally slower than arrays and List<T> due to dynamic resizing and boxing/unboxing overhead.
- **Declaration Example:**

Code

```
using System.Collections;
ArrayList myArrayList = new ArrayList();
```

```
  myArrayList.Add(10);
  myArrayList.Add("Hello");
```

3. List<T> (Generic List):

- **Dynamic Size:** Similar to ArrayList, its size can change dynamically.
- **Homogeneous (Type-Safe):** Stores elements of a single, specified data type (defined by the generic type parameter T). This provides type safety at compile time.
- **Generic:** Part of System.Collections.Generic namespace.
- **Performance:** Offers good performance, generally better than ArrayList because it avoids boxing/unboxing for value types and provides optimized resizing.
- **Declaration Example:**

Code

```
  using System.Collections.Generic;
  List<string> names = new List<string>();
  names.Add("Alice");
  names.Add("Bob");
```

LINQ

LINQ, or Language Integrated Query, is a powerful feature in C# that integrates query capabilities directly into the language itself. It provides a consistent model for querying and manipulating data from various data sources using a syntax similar to SQL, but within your LINQ queries can be written in two main forms:

- **Query Syntax:** Resembles SQL queries (e.g., from item in collection where condition select item).
- **Method Syntax:** Uses extension methods chained together (e.g., collection.Where(item => condition).Select(item => item)).

the compiler always converts the query syntax in method syntax at compile time

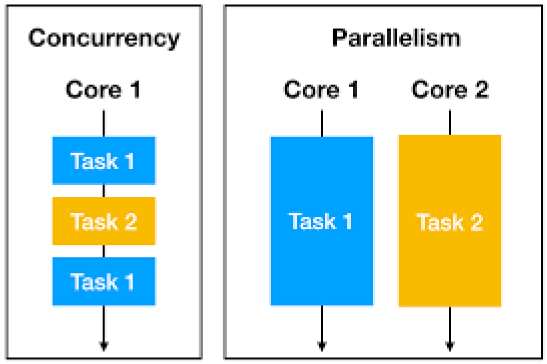
literal verbatim and interpolation
In C#, "literal," "verbatim," and "interpolation" refer to distinct ways of defining and constructing string values.

Literal String.
A standard string literal in C# is a sequence of characters enclosed in double quotes ("). Special characters within these literals, such as backslashes (` \ `) or double quotes, require "escaping" using a backslash.
Code
```
  string path = "C:\\Program Files\\My Application\\";
  string quote = "He said, \"Hello!\"";
```

Verbatim string literal.
A verbatim string literal is prefixed with an @ symbol. This tells the compiler to interpret the characters within the string literally, ignoring escape sequences. This is particularly useful for file paths or strings containing many backslashes or special characters.
Code
```
  string verbatimPath = @"C:\Program Files\My Application\";
  string multiLineText = @"This is a multi-line
  string literal.";
```
To include a double quote within a verbatim string, you must use two double quotes (""). string interpolation.

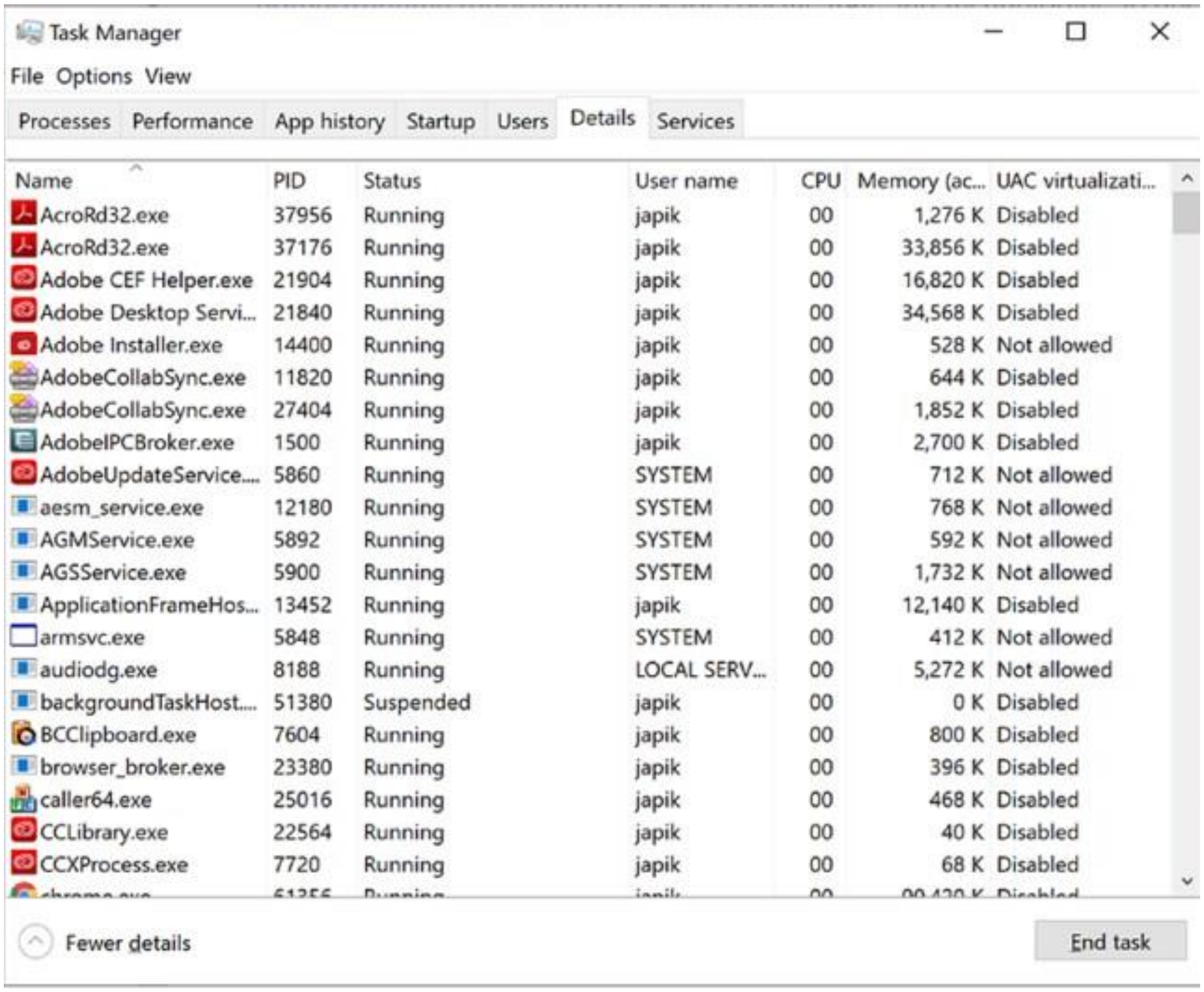
String interpolation, introduced in C# 6, provides a more readable and concise way to embed expressions or variables directly within a string literal. An interpolated string is prefixed with a $ symbol. The expressions to be evaluated are placed within curly braces ({}).
Code
```
  string name = "Alice";
  int age = 30;
  string greeting = $"Hello, {name}! You are {age} years old.";
```
String interpolation can be combined with verbatim strings by using both $ and @ prefixes (e.g., $@ or @$). This allows for literal interpretation of characters while also enabling expression embedding.
Code
```
  string folder = "Reports";
  string fileName = "Summary.txt";
  string interpolatedVerbatimPath =
$@"{Environment.CurrentDirectory}\{folder}\{fileName}";
```

Parallelism vs Thread
Parallelism refers to the ability to execute multiple tasks at the same time, often using multiple CPU cores.
Threading is a way to achieve concurrency within a single process, where multiple threads can run concurrently on a single processor or across multiple processors
the system switches between tasks to give the illusion of simultaneous execution.



A **process** is an instance of a program that is running on a computer. It is an independent entity, with its own memory space, code, data, and system resources. When you open a program like a web browser, an operating system (OS) creates a process for it.
A **thread** is the smallest unit of execution within a process. Threads within the same process share the same memory space, which allows them to easily communicate and share data. However, this shared memory space also means that threads need to be carefully synchronized to avoid conflicts and ensure data integrity.

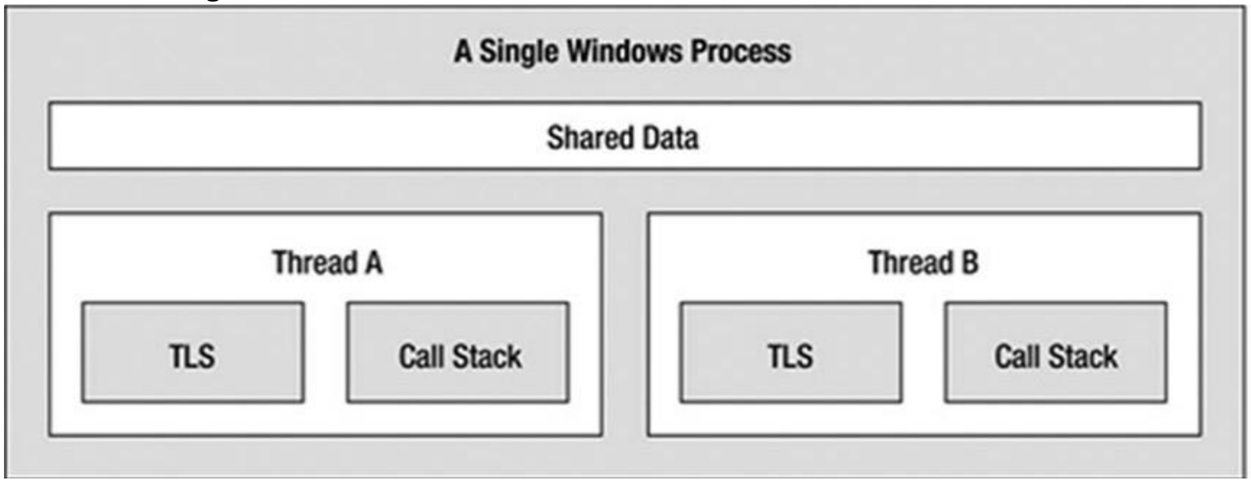
The concept of a “process” existed within Windows-based operating systems well before the release of the .NET/.NET Core platforms. In simple terms, a process is a running program. However, formally speaking, a process is an operating system–level concept used to describe a set of resources (such as external code libraries and the primary thread) and the necessary memory allocations used by a running application. For each .NET Core application loaded into memory, the OS creates a separate and isolated process for use during its lifetime.
Every Windows process is assigned a unique process identifier (PID) and may be independently loaded and unloaded by the OS as necessary (as well as programmatically). As you might be aware, the Processes tab of the Windows Task Manager utility (activated via the Ctrl+Shift+Esc keystroke combination on Windows) allows you to view various statistics regarding the processes running on a given machine. The Details tab allows you to view the assigned PID and image name (see image).


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	PID	Status	User name	CPU	Memory (ac...	UAC virtualizati...
AcroRd32.exe	37956	Running	japik	00	1,276 K	Disabled
AcroRd32.exe	37176	Running	japik	00	33,856 K	Disabled
Adobe CEF Helper.exe	21904	Running	japik	00	16,820 K	Disabled
Adobe Desktop Servi...	21840	Running	japik	00	34,568 K	Disabled
Adobe Installer.exe	14400	Running	japik	00	528 K	Not allowed
AdobeCollabSync.exe	11820	Running	japik	00	644 K	Disabled
AdobeCollabSync.exe	27404	Running	japik	00	1,852 K	Disabled
AdobeIPCBroker.exe	1500	Running	japik	00	2,700 K	Disabled
AdobeUpdateService....	5860	Running	SYSTEM	00	712 K	Not allowed
aesm_service.exe	12180	Running	SYSTEM	00	768 K	Not allowed
AGMService.exe	5892	Running	SYSTEM	00	592 K	Not allowed
AGSService.exe	5900	Running	SYSTEM	00	1,732 K	Not allowed
ApplicationFrameHos...	13452	Running	japik	00	12,140 K	Disabled
armsvc.exe	5848	Running	SYSTEM	00	412 K	Not allowed
audiodg.exe	8188	Running	LOCAL SERV...	00	5,272 K	Not allowed
backgroundTaskHost....	51380	Suspended	japik	00	0 K	Disabled
BCClipboard.exe	7604	Running	japik	00	800 K	Disabled
browser_broker.exe	23380	Running	japik	00	396 K	Disabled
caller64.exe	25016	Running	japik	00	468 K	Disabled
CCLibrary.exe	22564	Running	japik	00	40 K	Disabled
CCXProcess.exe	7720	Running	japik	00	68 K	Disabled

The Role of Threads
Every Windows process contains an initial "thread" (or the main thread) that functions as the entry point for the application. Chapter 15 examines the details of building multithreaded applications under the .NET Core platform; however, to facilitate the topics presented here, you need a few working definitions. First, a thread is a path of execution within a process. Formally speaking, the first thread created by a process's entry point is termed the primary thread(or the main thread). Any .NET Core program (console application, Windows service, WPF application, etc.) marks its entry point with the Main() method. When this method is invoked, the primary thread is created automatically Processes that contain a single primary thread of execution are intrinsically thread-safe, given the fact that there is only one thread that can access the data in the application at a given time. However, a single-threaded process (especially one that is GUI based) will often appear a bit unresponsive to the user if this single thread is performing a complex operation (such as printing out a lengthy text file, performing a mathematically intensive calculation, or attempting to connect to a remote server located thousands of miles away).

Given this potential drawback of single-threaded applications, the operating systems that are supported by .NET Core (as well as the .NET Core platform) make it possible for the primary thread to spawn additional secondary threads (also termed worker threads) using a handful of API functions such as CreateThread.
Each thread (primary or secondary) becomes a unique path of execution in the process and has concurrent access to all shared points of data within the process. As you might have guessed, developers typically create additional threads to help improve the program's overall responsiveness. Multithreaded processes provide the illusion that numerous activities are happening at more or less the same time. For example, an application may spawn a worker thread to perform a labor-intensive unit of work (again, such as printing a large text file). As this secondary thread is churning away, the main thread is still responsive to user input, which gives the entire process the potential of delivering greater performance. However, this may not actually be the case: using too many threads in a single process can actually degrade performance, as the CPU must switch between the active threads in the process (which takes time). On some machines, multithreading is most commonly an illusion provided by the OS. Machines that host a single (non-hyperthreaded) CPU do not have the ability to literally handle multiple threads at the same time. Rather, a single CPU will execute one thread for a unit of time (called a time slice) based in part on the thread's priority level. When a thread's time slice is up, the existing thread is suspended to allow another thread to perform its business. For a thread to remember what was happening before it was kicked out of the way, each thread is given the ability to write to Thread Local Storage (TLS) and is provided with a separate call stack, as illustrated in Figure



If the subject of threads is new to you, don't sweat the details. At this point, just remember that a thread is a unique path of execution within a Windows process. Every process has a primary thread (created via the executable's entry point) and may contain additional threads that have been programmatically created.

thread was defined as a path of execution within an executable application. While many .NET Core applications can live happy and productive single-threaded lives, an assembly's primary thread (spawned by the runtime when Main() executes) may create secondary threads of execution at any time to perform additional units of work. By creating additional threads, you can build more responsive (but not necessarily faster executing on single-core machines) applications. The System.Threading namespace was released with .NET 1.0 and offers one approach to build multithreaded applications. The Thread class is perhaps the core type, as it represents a given thread. If you want to programmatically obtain a reference to the thread currently executing a given member, simply call the static Thread.CurrentThread property, like so:

ThreadPool

The next thread-centric topic you will examine in this chapter is the role of the runtime thread pool. There is a cost with starting a new thread, so for purposes of efficiency, the thread pool holds onto created (but inactive) threads until needed. To allow you to interact with this pool of waiting threads, the System. Threading namespace provides the ThreadPool class type

Asynchrony— This means that your program performs non-blocking operations. For example, it can initiate a request for a remote resource via HTTP and then go on to do some other task while it waits for the response to be received. It's a bit like when you send an email and then go on with your life without waiting for a response.

Parallelism— This means that your program leverages the hardware of multi-core machines to execute tasks at the same time by breaking up work into tasks, each of which is executed on a separate core. It's a bit like singing in the shower: you're actually doing two things at exactly the same time.

Multithreading— This is a software implementation allowing different threads to be executed concurrently. A multithreaded program appears to be doing several things at the same time even when it's running on a single-core machine. This is a bit like chatting with different people through various IM windows; although you're actually switching back and forth, the net result is that you're having multiple conversations at the same time.

RESTful API
An API is considered RESTful when it follow the architectural principles of Representational State Transfer (REST). These principles, also known as constraints, guide the design of scalable, efficient, and maintainable web services.

The key elements that define a RESTful API include:

- **Client-Server Architecture:**
  There is a clear separation between the client and the server, allowing them to evolve independently. The client is responsible for the user interface and user experience, while the server handles data storage and processing.
- **Statelessness:**
  Each request from the client to the server must contain all the information necessary to understand and fulfill that request. The server does not store any client state between requests, making the API more scalable and reliable.
- **Cacheability:**
  Resources can be cached on the client or server side to improve performance and scalability. Server responses should include information about whether caching is allowed for the delivered resource.
- **Uniform Interface:**
  This is a fundamental principle that simplifies the overall system architecture. It includes:
  - **Resource Identification in Requests:** Resources are identified by unique URIs (Uniform Resource Identifiers).
  - **Resource Manipulation Through Representations:** Clients interact with resources by manipulating their representations (e.g., JSON, XML).
  - **Self-Descriptive Messages:** Messages exchanged between client and server are self-contained and provide enough information to be understood.
  - **Hypermedia as the Engine of Application State (HATEOAS):** The server provides hypermedia links within its responses, guiding the client on how to transition to different application states.
- **Layered System:**
  The architecture can be composed of multiple layers (e.g., proxies, load balancers), and the client should not be able to tell if it's connected directly to the end server or to an intermediary.
- **Code-on-Demand (Optional):**

> Servers can temporarily extend or customize the functionality of a client by transferring executable code (e.g., JavaScript applets). This is an optional constraint.

Method Hiding, Shadowing, and Overriding in C# Explained with Examples

Method hiding, shadowing, and overriding are important concepts in object-oriented programming using C#.

They involve the use of virtual, override, and new keywords to define and implement methods in derived classes that are related to methods in base classes. In this blog post, we will explore these concepts in detail and provide examples to illustrate how they work in C#.

Method Hiding:

- Method hiding is a concept where a derived class defines a new method with the same name as a method in the base class, effectively hiding the base class method.
- Method hiding is achieved by using the new keyword in the derived class method.
- The base class method should be marked as virtual to be hidden by the derived class method.
- The derived class method must have the same name as the base class method, but it can have a different return type or method signature.
- Example:

```
public class Animal
{
    public virtual void MakeSound()
    {
        Console.WriteLine("Animal makes a sound.");
    }
}

public class Cat : Animal
{
    public new void MakeSound()
    {
        Console.WriteLine("Cat makes a sound.");
    }
}
```

```
public class Program
{
    static void Main()
    {
       // Method Hiding
        Animal animal1 = new Animal();
        Animal cat1 = new Cat();
        Cat animal2 = new Animal(); // Compilation error - cannot implicitly
convert Animal to Cat
        Cat cat2 = new Cat();

        animal1.MakeSound(); // Output: "Animal makes a sound."
        cat1.MakeSound();    // Output: "Animal makes a sound." (hiding)
        cat2.MakeSound();    // Output: "Cat makes a sound."
    }
}
```

Method Shadowing:

- Method shadowing is similar to method hiding, where a derived class defines a new method with the same name as a method in the base class.
- However, in method shadowing, the base class method is not marked as virtual , and the derived class method does not use the new keyword.
- This can lead to unexpected behavior as the base class method may still be invoked when calling the method on an object of the derived class.
- Example:

```
public class Animal
{
    public void MakeSound()
    {
        Console.WriteLine("Animal makes a sound.");
    }
}

public class Cat : Animal
{
    public void MakeSound()
    {
        Console.WriteLine("Cat makes a sound.");
    }
}


public class Program
{
    static void Main()
    {
        // Method Shadowing
        Animal animal1 = new Animal();
        Animal cat1 = new Cat();
        Cat animal2 = new Animal(); // Compilation error - cannot implicitly
convert Animal to Cat
        Cat cat2 = new Cat();

        animal1.MakeSound(); // Output: "Animal makes a sound."
        cat1.MakeSound();    // Output: "Animal makes a sound." (shadowing)
        cat2.MakeSound();    // Output: "Cat makes a sound."
    }
}
```

Method Overriding:

- Method overriding is a concept where a derived class provides a new implementation of a base class method, and the new implementation is used when calling the method on an object of the derived class.
- Method overriding is achieved by using the override keyword in the derived class method.
- The base class method should be marked as virtual and the derived class method must have the same name, return type, and method signature.
- Example:

```
public class Animal
{
    public virtual void MakeSound()
    {
        Console.WriteLine("Animal makes a sound.");
    }
}
public class Cat : Animal
```

```
{
    public override void MakeSound()
    {
        Console.WriteLine("Cat makes a sound.");
    }
}

public class Program
{
    static void Main()
    {
        // Method Overriding
        Animal animal1 = new Animal();
        Cat animal2 = new Animal(); // Compilation error - cannot implicitly
convert Animal to Cat
        Animal cat1 = new Cat();
        Cat cat2 = new Cat();

        animal1.MakeSound(); // Output: "Animal makes a sound."
        cat1.MakeSound();    // Output: "Cat makes a sound." (overriding)
        cat2.MakeSound();    // Output: "Cat makes a sound."
    }
}
```

Differences between Method Hiding, Shadowing, and Overriding:

1. **Method Hiding**:

- Derived class defines a new method with the same name as a method in the base class, effectively hiding the base class method.
- new keyword is used in the derived class method.
- Base class method should be marked as virtual or override.
- Derived class method can have a different return type or method signature.

2. Method Shadowing:

- Derived class defines a new method with the same name as a method in the base class, but the base class method is **not marked** as virtual or override.
- Base class method may still be invoked when calling the method on an object of the derived class.
- No new keyword is used in the derived class method.

3. Method Overriding:

- Derived class provides a new implementation of a base class method.
- override keyword is used in the derived class method.
- Base class method should be marked as virtual, abstract, or override.
- Derived class method must have the same name, return type, and method signature as the base class method.

In conclusion, method hiding, shadowing, and overriding are important concepts in C# that allow for customization and extension of base class methods in derived classes. By following the proper guidelines for creating objects and implementing these concepts, you can ensure correct and expected behavior in your object-oriented programming code.

I hope this detailed blog post on Method hiding, shadowing and overriding in C# with examples and best practices has been helpful to you. Remember to apply these concepts wisely in your code and make use of the code examples provided to enhance your understanding. Happy coding!