# 1. Basic Concepts

## C# Overview

C# is a programming language:

Language Features: C# is a modern, object-oriented, and type-safe programming language. It offers features like automatic garbage collection, exception handling, generics, LINQ, async programming, and more.

Use Cases: While it's commonly used for developing Windows applications, it can also be used for web, mobile, and cloud applications, especially with the .NET ecosystem.

Development Environment: C# is most commonly used with integrated development environments (IDEs) like Visual Studio, which provide extensive tools for development, debugging, and testing.

## Value types

- Both value types and reference types are passed by value by default. However, the value of a reference type variable is the 'pointer' to a memory address on the heap.

- Value types are typically stored on the stack rather than the heap. This means they are allocated and deallocated faster than reference types, which are stored on the heap.

- Value types do not have a null value, because they always have a default value. For example, the default value of int is 0.

- Value types can be converted to reference types using box operator, which creates a new object on the heap that contains a copy of the value. This is useful when a value type needs to be stored in a collection of reference types or when it needs to be passed as an argument to a method that expects a reference type.

- Include simple types, such as int, float and bool as well as enumerations and structures.

## Reference types

- Reference types are stored on the heap, rather than the stack. This means they are allocated and deallocated by the garbage collector, which monitors the heap and reclaims memory that is no longer being used by the application.

- Reference types can have a null value, which means that they do not refer to any object. This is different from value types, which always have a default value.

- Both value types and reference types are passed by value by default. However, the value of a reference type variable is the 'pointer' to a memory address on the heap.

- Reference types in .NET include class types, such as String and Array as well as interface types and delegate types.

## Value parameters

- The most common type of parameter in .NET. They are used to pass a copy of the argument value to the method or function

- Any changes made to the parameter inside the method or function does not affect the original argument

- Value parameters are passed by value, which means that the argument is copied into the parameter

## Reference parameters

- They are passed by value which is a reference to an address on the heap

- The method or function receives a reference to the orginal argument, rather than a copy of its value

- Any changes made to the parameter inside the method or function are reflected in the original argument

## Mutable types

- A mutable type is a type whose state can be modified after it is created. An immutable type, on the other hand, is a type whose state cannot be change after it is created.

- Some examples of mutable types in .NET include classes like System.StringBuilder and System.Data.Dataset, which allows you to modify their contents after they are created. For example, you can use the Append() method of System.StringBuilder to add more characters to a string, or you can use the Add() method of System.Data.DataSet to add new rows to a dataset.

## Immutable types

- System.String is an immutable type, because once you create a string, you cannot change its contents. If you want to modify a string, you must create a new string with desired modifications

- If you create a reference to a string, and then modify the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified.The following code illustrates this behaviour:

```
string str1 = "Hello ";
string str2 = str1;
str1 += "World";

System.Console.WriteLine(str2);
//Output: Hello
```

In general it is considered good practice to use immutable types whenever possible, because they can prevent bugs and make your code easier to reason about

## Mutability example - Dictionaries

- Dictionaries are mutable which means their state can be modified after they are created. This means you can add, or modify the elements of a dictionary after it is created.
- For example, you can use the Add() method of the System.Collections.Generic.Dictionary<TKey, TValue> class to add new key-value pairs to a dictionary, or you can use the Remove() method to remove elements from a dictionary

- While mutable dictionaries can be useful in some cases, it is generally considered good pratice to use immutable data structures wherever possible.

- In .NET, you can use the System.Immutable.ImmutableDictionary<TKey, TValue> class to create an immutable dictionary.

## Float vs Double

- Float and double data types are both used to represent floating-point numbers

- A float is a single-precision 32-bit floating-point number. A double is a double-precision 64-bit floating point number

- Double has 2x the precision of float. In general a double has 15 decimals of precision, while a float has 7.

- The maximum value of a float is about 3e38, but double is about 1.7e308, so using a float can hit "infinity" (ie. a spcial floating point number) much more easiliy than double for something simple like computing the factorial of 60

## Out keyword

To use an out parameter, both the method definition and the calling method must explicitly use the out keyword. For example:

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

Variables passed as out arguments don't have to be initialised before being passed in a method call. However, the called method is required to assign a value before the method returns.

Deconstruct methods declare their parameters with the out modifier to return multiple values. Other methods can return value tuples for multiple return values.

You can declare a variable in a separate statement before you pass it as an out argument. You can also declare the out variable in the argument list of the method call, rather than in a separate variable declaration. out variable declarations produce more compact, readable code, and also prevent you from inadvertently assigning a value to the variable before the method call. The following example defines the number variable in the call to the Int32.TryParse method.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
```

//  Converted '1640' to 1640

You can also declare an implicitly typed local variable.

## Null-coalescing operator

The **null-coalescing operator** in .NET, represented by `??`, is a binary operator used in C# and other .NET languages to simplify null checks and provide default values for nullable expressions. It allows developers to write more concise and readable code by handling `null` values efficiently.

## Syntax

```
expression1 ?? expression2
```

- **`expression1`**: The first operand, which may be `null`.
- **`expression2`**: The second operand, which is returned if `expression1` is `null`.

```
string name = null;
string displayName = name ?? "Guest";

Console.WriteLine(displayName); // Output: Guest
```

## Exception handling

- The.NET exception class is the base class for all exceptions in the .NET framework. It is defined in the 'System' namespace and provides common functionality for handling exceptions

- The exception class provides several properties that can be used to obtain information about an exception, such as the message, the stack trace, and the inner exception. It also provides a 'ToString' methods that can be used to generate a string representation of the exception, including the message and stack trace.

- The exception class is not typically used directly, but it is the base class for many other exception classes in the .NET framework. For example, the 'ArgumentException' class is derived from 'Exception' and is used to represent an error in the arguments passed to a method

- 'FileNotFoundException' class is also derived from 'Exception' and is used to represent an error when a file cannot be found

- Finally block is where you write code that should execute whether or not you have an exception

**Casting**

- Involves moving from one datatype to another. Like integer to double
- Casting is generally meant for compatible types

**Implicit casting**

*Int i = 10;*
*Double d = i; // implicit casting*

When moving from a lower data type to a higher data type, you will normally have implicit casting

**Explicit Casting**

*Double d1 = 100.23*
*Int y = (int)d1;*

When moving from a higher data type to a lower data type, you will need explicit casting

Consequences of explicit casting can be data type. In above example, since int doesn't support decimals, you will have data loss.

**Overview of Casting in Object-Oriented Programming (OOP)**

Casting in object-oriented programming allows you to treat an object as if it were of another type within its inheritance hierarchy. This is a fundamental concept that enables polymorphism, code flexibility, and reusability. Casting is generally categorized into two types:

1. **Upcasting**: Casting a derived class object to a base class type.
2. **Downcasting**: Casting a base class reference back to a derived class type.

Below, we'll explore both upcasting and downcasting, including the examples previously provided.

---

## 1. Upcasting

**What is Upcasting?**

Upcasting is the process of treating an object of a derived class as an object of its base class. Since a derived class inherits from the base class, it inherently possesses all the characteristics of the base class. Therefore, upcasting is safe and often implicit, meaning you don't need to explicitly write the cast in code.

**Benefits of Upcasting:**

- **Polymorphism**: Allows you to write code that works on the base class type, which can handle any of its derived types.
- **Code Reusability and Flexibility**: Enables methods and collections to work with base class types, making your code more general and reusable.
- **Simplified Object Management**: You can manage a group of objects of various derived types uniformly through base class references.

**Example of Upcasting:**

Suppose we have a base class `Mammal` and derived classes `Dog` and `Cat`.

```
class Mammal
```

```
{
    public virtual void Feed()
    {
        Console.WriteLine("The mammal is being fed.");
    }
}

class Dog : Mammal
{
    public override void Feed()
    {
        Console.WriteLine("The dog is eating.");
    }
}

class Cat : Mammal
{
    public override void Feed()
    {
        Console.WriteLine("The cat is eating.");
    }
}

class Program
{
    public static void FeedMammal(Mammal mammal)
    {
        mammal.Feed();  // Calls the appropriate Feed method based on the actual
object type
    }

    static void Main(string[] args)
    {
        Dog dog = new Dog();
        Cat cat = new Cat();

        // Upcasting Dog and Cat to Mammal
        Mammal mammalDog = dog;  // Implicit upcast
        Mammal mammalCat = cat;  // Implicit upcast

        // Call a generic method that works with any Mammal
        FeedMammal(mammalDog);  // Outputs: "The dog is eating."
        FeedMammal(mammalCat);  // Outputs: "The cat is eating."
    }
}
```

**Explanation:**

- **Polymorphism in Action**: Even though `mammalDog` and `mammalCat` are of type `Mammal`, the overridden `Feed()` method in `Dog` and `Cat` is called due to polymorphism.

- **Generic Programming**: The `FeedMammal` method can accept any `Mammal`, making it reusable and flexible.
- **Simplifies Code Maintenance**: Adding new types of mammals (e.g., `Elephant`, `Whale`) requires minimal changes to existing code.

---

## 2. Downcasting

**What is Downcasting?**

Downcasting is the process of converting a base class reference back to a derived class type. This is required when you need to access methods or properties that are specific to the derived class. Downcasting can be unsafe if not done carefully, and it often requires an explicit cast.

**Benefits of Downcasting:**

- **Access to Specialized Members**: Allows you to use methods and properties that are specific to the derived class.
- **Flexibility in Code**: Enables handling of objects in a generic manner while still being able to perform specific operations when needed.

**Example of Downcasting:**

Continuing with the previous classes, let's say the `Dog` class has a method `GiveTrainingCommand()` which is not present in the base class `Animal`.

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

class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("The dog barks.");
    }

    public void GiveTrainingCommand()
    {
        Console.WriteLine("The dog is being trained to sit.");
    }
}

class Program
{
    public static void ManageAnimal(Animal animal)
```

```
    {
        animal.Speak();  // Calls the overridden Speak method

        // Downcasting to Dog to access Dog-specific method
        if (animal is Dog dog)
        {
            dog.GiveTrainingCommand();  // Dog-specific behavior
        }
        else
        {
            Console.WriteLine("This animal cannot be trained with dog-specific
commands.");
        }
    }

    static void Main(string[] args)
    {
        Animal genericAnimal = new Animal();
        Animal dog = new Dog();

        // Managing a generic animal
        ManageAnimal(genericAnimal);
        // Output:
        // The animal makes a sound.
        // This animal cannot be trained with dog-specific commands.

        // Managing a dog, downcasting to access Dog-specific behavior
        ManageAnimal(dog);
        // Output:
        // The dog barks.
        // The dog is being trained to sit.
    }
}
```

**Explanation:**

- **Type Checking with Pattern Matching**: The `if (animal is Dog dog)` statement checks if `animal` is of type `Dog` and, if so, casts it to `dog`.
- **Access to Derived Class Methods**: By downcasting, we can access `GiveTrainingCommand()`, which is specific to `Dog`.
- **Generic Handling with Specific Actions**: The `ManageAnimal` method can handle any `Animal`, but also performs specific actions when the animal is a `Dog`.

---

## Summary of Casting

- **Upcasting**:
    - **Safe and Implicit**: No explicit cast required.

    - **Uses**: To treat a derived class object as an instance of its base class.
    - **Benefits**:
        - Code generalization.
        - Polymorphism.
        - Reusability.
        - Simplified object management.
- **Downcasting**:
    - **Requires Explicit Cast or Type Check**: Use `is` or `as` to ensure safety.
    - **Uses**: To access members of the derived class that are not available in the base class.
    - **Benefits**:
        - Access to specialized behavior.
        - Flexibility to perform specific operations when needed.

## Key Points to Remember

- **Casting Follows the Inheritance Chain**: You can only cast between types that are in the same inheritance hierarchy.
- **Upcasting is Always Safe**: Because a derived class object is also an instance of its base class.
- **Downcasting Requires Caution**:
    - Use type checking (`is`, `as`) to prevent runtime exceptions.
    - Ensure that the object is actually an instance of the derived class.

## Practical Applications

**Collections of Base Type**: You can store objects of various derived types in a collection of base class type, and upcasting makes this seamless.

```
List<Animal> animals = new List<Animal>
{
    new Dog(),
    new Cat(),
    new Animal()
};

foreach (var animal in animals)
{
    animal.Speak();  // Polymorphic call
}
```

- 
- **Generic Methods**: Methods that operate on the base class can work with any derived class objects, promoting code reuse.
- **Dynamic Behavior**: Polymorphism allows methods to behave differently based on the actual object's type, even when accessed through a base class reference.

---

By understanding and utilizing both upcasting and downcasting, you can write flexible, maintainable, and reusable code that leverages the full power of object-oriented programming.

# 2. Object-Oriented Programming

## Key points

- User defined data types that represent state and behaviour of an object

- Classes are reference types that hold the object created dynamically on the heap. Programmer specifies the accessibility of a class, method, or property

- The ultimate base type of all classes is object Every type inherits from Object. Classes, structs, enums (which are just structs), and delegates (which are just classes).

- Default access modifier is Internal

- Default access modifiers of methods variables is private

## Type of Classes

**Abstract**

- Cannot be instantiated. Must be inherited

- Contains both abstract and non-abstract methods. Abstract methods don't have implementation and are overridden

- A class in C# can only inherit from one abstract class (You can only inherit from a single class in .Net. It is however possible to implement multiple interfaces)

- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

**Partial**

- Allows dividing a class's properties, methods and events into a multiple source files and at compile time their files are compiled into a single class

- If you seal a specific part of a partial class then the entire class is sealed

- If a partial class inherits from a base class, it must be declared in only one part of the partial class.

- All parts of the partial class will inherit from the base class, as they are considered as a single class during compilation.

- Similarly, if a partial class is meant to be a base class, all of its parts collectively represent the base class.

**Static**

- Cannot be instantiated such that class members can be called directly using their class name

- Inside a static class, only static members are allowed

- Static class cannot be inherited

**Sealed**

- Cannot be inherited by other classes. They are used to prevent other classes from extending or modifying behaviour.

## Access modifiers

**Public**

- The members that are declared with the 'public' access modifier can be accessed from anywhere in the program

**Private**

- The members that are declared with the 'private' access modifier can only be accessed within the same class

**Protected**

- The members that are declared with the 'protected' access modifier can be accessed within the same class and by derived classes

**Internal**

- The members that are declared with the internal access modifier can be accessed within the assembly it is declared but not in other assemblies

## Virtual methods

- A virtual method is a class method that offers functionality to the programmer to override a method in the derived class (first created in base class) that has the same signature

- Virtual methods are mainly used to perform polymorphism in the OOP environment

- A virtual method can be created in the base class by using the "virtual" keyword and the same method can be overridden in the derived class by using the "override keyword"

- It's optional to override a virtual method in the derived class

Abstract class vs interface

**When should I use an abstract class?**

Good choice if you are bringing into account the inheritance concept because it provides common base class implementation to the derived classes

- Also good if you want to declare non-public members. In an interface, all methods must be public

- If you want to declare new methods in the future, then it is great to go with an abstract class. If you add new methods to the interface, then all of the classes that are already implemented in the interface will have to be changed in order to implement these new methods

- If you want to create multiple versions of your component, then go with abstract class. They provide a simple and easy way to version your components. When you update the base class, all of the inheriting classes would be automatically updated with the change. Interfaces, on the other hand can't be changed once these are created. If you want a new version of your interface, then you must create a new interface.

**When should I use an interface?**

- If you are creating functionality that will be useful across a wide range of objects, then use an interface. Abstract classes should be used for objects that are closely related. But the interfaces are best suited for providing common functionality to unrelated causes.

- Interfaces are good choice if you think API won't be changing for a while

- If we are going to design small, concise bits of functionality, then you must use interfaces. But if you are designing large functional units, then you should use an abstract class

## Structs

Value types that are typically used for small, simple objects that have a short lifespan and represent a single value or a closely related set of data. Understanding when to use structs instead of classes, which are reference types, is crucial for writing efficient and effective C# code. Here are some guidelines for when you might choose to use structs:

**Small and Lightweight Objects:**

Structs are best suited for small data structures. The .NET guidelines suggest using structs for objects that are smaller than 16 bytes. Larger structs can lead to performance issues due to the cost of copying them.

**Immutable Data**:

Structs are a good choice when creating immutable types. Once a struct is created, its data should not be changed. This immutability can help prevent bugs and makes the code easier to understand.

**Value Semantics:**

Use structs when you need value semantics, where each instance is independent and modifications to one instance do not affect others. This is in contrast to reference types, where different variables can reference the same object.

# 3. Advanced C# Features

## Delegates

**A bit of help with delegates and events:**
https://www.reddit.com/r/csharp/comments/s1bonr/comment/hs857vs/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1&utm_content=share_button

- A delegate is a type that holds a reference to a method. Declared with a signature that shows the return type and parameters for the methods it references, it can hold references only to methods that match its signature

```
// Define a delegate type for a method that takes an integer and returns a string

public delegate string IntToString(int i);

// Define two methods that match the delegate signature

public static string IntToBinaryString(int i)

{

    return Convert.ToString(i, 2);

}

public static string IntToHexString(int i)

{

    return Convert.ToString(i, 16);

}

// Define a method that takes a delegate as an argument

public static void PrintIntAsString(IntToString convert, int i)

{

    Console.WriteLine(convert(i));

}

// Use the PrintIntAsString method with the two methods defined above

public static void Main()

{
```

```
    PrintIntAsString(IntToBinaryString, 10);

    PrintIntAsString(IntToHexString, 10);

}
```

- In the context of events, a delegate is used to specify the signature of the event handling method.

## Events

- To respond to an event, you define an event handler method in the event receiver.

- This method must match the signature type of the delegate for the event your're handling

- In the event handler, you perform the actions that are required when the event is raised, such as collecting user input after the user clicks a button

- To receive notifications when the event occurs your event handler method must subscribe to the event

- The following example shows an event handler method named c_ThresholdReached that matches the signature for the EventHandler delegate. The method subscribes to the ThresholdReached event:

```
class ProgramTwo

{

    static void Main()

    {

        var c = new Counter();

        c.ThresholdReached += c_ThresholdReached;



        // provide remaining implementation for the class

    }

    static void c_ThresholdReached(object sender, EventArgs e)

    {

        Console.WriteLine("The threshold was reached.");

    }

}
```

## Boxing

- Refers to the process of converting a value type (such as an integer or a boolean value) into a reference type

- This is done by creating a new object that contains the value of the original type, and storing a reference to this new object.

```
// Declare an integer value

int i = 5;

// Box the integer value

object o = i;
```

- In this example, the integer value 5 is boxed and stored in the object reference type o.

- This allows us to treat the value of i as an object, even though it is originally a value type.

- Boxing / unboxing affects performance because of jumping from stack to heap etc

## Unboxing

- Unboxing refers to the process of converting a reference type (an object) into a value type

- It allows you to access the value stored in an object and treat it like a normal value of the corresponding value type.

```
object myObject = 5;

int myInt = (int)myObject; // unboxing
```

- In this example, myObject is a reference type that contains the value 5.

- When we use (int) syntax to cast myObject to an int we are performing unboxing and acessing the value stored in myObject. The value is then stored in myInt which is a value type

## Boxing and unboxing used by non-generic collections

- Boxing and unboxing are concepts used with non-generic collections, which are collections that can store objects of any data type

- When you add a value to a non-generic collection, the value type is automatically boxed. This means that the value is converted to a reference type and stored in the collection as an object

- When you retrieve a value from the collection, it must be unboxed, which means that the reference type is converted back to a value type

```
// Create a non-generic collection of ints

ArrayList collection = new ArrayList();

// Add a value to the collection

int value = 42;

collection.Add(value);

// Retrieve the value from the collection

int unboxedValue = (int)collection[0];
```

- In this example, the int value is added to the ArrayList collection, which boxes the value and stores it as an object. When the value is retrieved from the collection, it is unboxed and converted back to an int.

- Boxing and unboxing are useful in certain situations - they can also have a negative impact on performance.

- Therefore, it's generally recommended to use generic collections whenever possible, as they can avoid the overhead of boxing and unboxing

# 4. Data Structures and Collections

## Collections

- System.Collections namespace

- You often want to create and manage a group of related objects. There are two ways to group objects: by creating arrays of objects and by creating collections of objects.

- Arrays are most useful for creating and working with a fixed number of strongly typed objects

- Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change

- For some collections, you can assign a key to any object that you put in the collection so that you can quickly retrieve the object by using the key

- Common collection classes are: List<T>, Dictionary<TKey, TValue> etc

- There is almost 0 reason to use non generic collections. Generics were introduced in .net 2.0 there is very little code that exists now that can't take advantage of generic collections

## Generics

- System.Collections.Generic contains classes and interfaces that define generic collections, which allows users to create strongly typed collections that provide a better type safety and performance than non-generic strong typed collections

- Examples of System.Collections.Generic classes include: List<T>, Dictionary<TKey, TValue>

- Generics introduces the concept of type parameters to .NET, which makes it possible to design classes and methods that defer specification of one of more types until the class or method is declared and instantiated by client code
- For example, by using a generic type of paramater T, you can write a single class that other client code can use without incurring the cost of runtime casts or boxing operations

## Array

- Part of System.Object namespace

- Can store multiple variables of the same type in an array data structure

- If you want to store elements of any type, you can specify object as its type

- In the unified type system of C#, all types, predefined and user-defined inherit directly from Object

- When you initialize a C# array, the .NET runtime reserves a block of memory sufficient to hold the elements. It then stores the elements of the array sequentially within that block of memory meaning they are very efficient

- They are implemented in runtime which is why they get special syntax that no other type has

## List

- The List<T> class is a sequentialy and dynamically resizable list of items. Under the hood, List<T> is based on an array
- The List<T> class has three main fields:
1. T[] _items is an internal array. The list is built on the base of this array
2. Int_size stores information about the number of items in the list
3. Int_version stores the version of the collection

## Dictionary

- Under the hood, a dictionary in .NET is implemented using a hash table. A hash table is a data structure that stores key-value pairs and provides fast lookups, inserts, and deletes of the values based on their keys.

- In a dictionary, the keys are used to compute a hash code, which is an integer value that represents the key. The hash code is then used to determine the index in the hash table where the value will be stored. This process is called hashing.

- When you look up a value in a dictionary, the key is hashed to compute its hash code, and then the value is looked up in the hash table at the index determined by the hash code. This allows the dictionary to quickly find the value associated with a given key.

- Inserting and deleting key-value pairs from a dictionary also involves computing the hash code of the key and using it to determine the index in the hash table where the value should be stored or removed.

- One of the benefits of using a hash table to implement a dictionary is that it provides fast lookups, inserts, and deletes of key-value pairs, with an average-case time complexity of O(1). This means that, on average, it takes a constant amount of time to perform these operations, regardless of the size of the dictionary.

## Concurrent Dictionary

- Represents a thread-safe collection of key-value pairs that can be accessed by multiple threads concurrently

- Without concurrentDictionary class, if we have to use Dictionary class with multiple threads, then we have to use locks to provide thread-safety which is often error-prone

## Performance

- To look up a key in a hash table: **O(1)**

- To look up key in a list using linear search: **O(N)**

- To look up key in an array using linear search: **O(N)** - array is faster than list since elements are stored continuously in memory

- When comparing the performance of searching through an array vs list, it is important to remember two types of memory:

- **Static memory:** The type of memory that is defined at compile time. This memory is reserved for a variable and cannot be changed at runtime

- **Dynamic memory:** The type of memory used at runtime. This memory space is also reserved for a variable, but in this case it can be modified at runtime

- List uses dynamic memory while array uses static memory

## IEnumerable Interface

- Represents a collection of objects that can be enumerated, or accessed one at a time. It is a generic interface, which means that it can be used with any data type

- IEnumerable is typically used when working with collections of objects, such as lists or arrays

- Allows you to write code that can iterate over the elements in the collection, without having to know the specific type of objects that the collection contains

- In the below example, the collection must implement the IEnumerable interface in order to use the foreach loop:

```
List<string> names = new List<string>() { "John", "Jane", "Bob" };

foreach (string name in names)
```

*{*

*Console.WriteLine(name);*

*}*

- In this example, the names list implements the IEnumerable interface, so it can be used with foreach

## GetEnumerator();

- Typically used in conjunction with the IEnumerable interface, which represents a collection of objects that can be enumerated

- The IEnumerator interface defines a single method called MoveNext, which is used to move to the next element in the collection. Also defines two properties, Current and Reset, which are used to access the current element in the collection and reset the enumerator to the beginning of the collection, respectively.

- Example of using the IEnumerator interface to iterate over a collection fo strings:

```
List<string> names = new List<string>() { "John", "Jane", "Bob" };

IEnumerator<string> enumerator = names.GetEnumerator();

while (enumerator.MoveNext())

{

    Console.WriteLine(enumerator.Current);

}
```

- Above, the names list implements the IEnumerable interface, so it can be used with the GetEnumerator method to obtain a IEnumerator object

- The IEnumerator object is then used in a while loop to iterate over the elements in the names list. The MoveNext() method is used to move to the next element in the collection, and the Current property is used to access the current element

- IEnumerator is an important concept in .NET programming because it allows you to write code that can iterate over a collection of objects in a generic and reusable way

## Array vs ArrayList

**- Arraylists are obsolete and should never be used in anything close to modern c# however here is comparison:**

- Array is fixed length. You can use resize to change length but it isn't very straight-forward

- Array is strongly-typed

- ArrayList is flexible in terms of number of elements. It is not strongly-typed

- Performance of array is better since there is no boxing / unboxing as it is strongly-typed

- When adding a value, boxing will take place - value type to reference type

- When retrieving a value, unboxing will take place

# 5. Memory Management

## Key points

- Garbage collection is a CLR process that keeps running continuously, automatically freeing up memory that is no longer used by the program

- The garbage collector is a background process that runs on a separate thread from the main program

- Managed resources are those that are pure .NET code and managed by the runtime and are under its direct control. Garbage collectors cannot collect objects created outside the CLR runtime.

- Unmanaged resources are those that are not. File handles, pinned memory, COM objects, database connections etc.

## Generations of the Garbage Collector

The .NET garbage collector (GC) uses a generational approach to manage memory more efficiently. This generational model is based on the observation that most objects are short-lived. To optimize performance, the GC organizes managed objects into three generations: Generation 0, Generation 1, and Generation 2. Each generation is collected separately, with different frequencies and optimizations.

1. **Generation 0 (Gen 0)**
2. **Generation 1 (Gen 1)**
3. **Generation 2 (Gen 2)**

## 1. Generation 0 (Gen 0)

- **Description**: Generation 0 is where new objects are allocated. It is the first generation that is considered for garbage collection.
- **Characteristics**:
    - Contains short-lived objects, such as temporary variables and short-lived instances.
    - Garbage collection happens most frequently in Generation 0.
    - When a Generation 0 collection occurs, objects that survive the collection (i.e., still referenced) are promoted to Generation 1.
- **Performance**:
    - Since most objects are short-lived, a Gen 0 collection is typically quick and efficient.
    - If the GC finds that many objects in Gen 0 are still alive, it may trigger a Gen 1 collection to move surviving objects to Gen 1.

## 2. Generation 1 (Gen 1)

- **Description**: Generation 1 acts as a buffer between the short-lived objects in Gen 0 and the long-lived objects in Gen 2.
- **Characteristics**:
    - Contains objects that survived a Generation 0 collection.
    - Objects in Generation 1 are considered to have medium lifetimes.
    - Less frequently collected than Generation 0 but more often than Generation 2.
    - When a Generation 1 collection occurs, objects that survive the collection are promoted to Generation 2.
- **Performance**:
    - Generation 1 collections are more expensive than Generation 0 collections but still relatively efficient.
    - Helps to filter out objects that are longer-lived but not yet ready to be promoted to Generation 2.

## 3. Generation 2 (Gen 2)

- **Description**: Generation 2 contains long-lived objects, such as objects that stay in memory for the duration of an application's lifetime (e.g., static data, caches, and large objects).
- **Characteristics**:
    - Objects that survive multiple collections in Generation 1 are promoted to Generation 2.
    - Generation 2 is collected the least frequently but is the most expensive to collect.
    - When a full garbage collection (also known as a **full GC**) is triggered, all three generations (0, 1, and 2) are collected.
- **Performance**:
    - Generation 2 collections are the most resource-intensive, as they involve a larger memory space and potentially many long-lived objects.
    - Full GC should be minimized because of the performance impact on the application.

## The Large Object Heap (LOH)

- **Description**: The Large Object Heap is a separate heap that stores large objects (e.g., arrays or objects larger than 85,000 bytes).
- **Characteristics**:
    - Large objects are always allocated directly in the Generation 2 heap.
    - The LOH is collected only during a Generation 2 collection.
    - LOH fragmentation can impact performance; .NET 4.5 and later versions have a feature called "Large Object Heap Compaction" to reduce fragmentation.

## Why Use Generations?

- **Optimization for Short-Lived Objects**: Most objects are short-lived (e.g., temporary variables), so collecting them more frequently reduces memory pressure quickly and efficiently.
- **Reduced Overhead**: By promoting only surviving objects to higher generations, the garbage collector avoids collecting the entire heap frequently, which would be costly.
- **Improved Performance**: Dividing memory into generations allows the garbage collector to focus on areas with high churn (e.g., Gen 0) while minimizing work in areas with stable data (e.g., Gen 2).

## Summary

- **Generation 0**: Short-lived objects; frequent, low-cost collections.

- **Generation 1**: Medium-lived objects; acts as a buffer between short-lived and long-lived objects.
- **Generation 2**: Long-lived objects; infrequent but expensive collections.
- **Large Object Heap (LOH)**: For large objects; collected with Gen 2.

This generational approach helps the .NET garbage collector balance performance and memory efficiency, providing a smooth and responsive experience for .NET applications.

—----------

In the context of garbage collection (GC) in .NET, the three service lifetimes—**Transient**, **Scoped**, and **Singleton**—affect when and how objects are cleaned up by the garbage collector. Here's a breakdown of each in this context:

## 1. Transient (`AddTransient`)

- **Behavior**: A new instance of the service is created each time it is requested from the dependency injection (DI) container.
- **Garbage Collection Context**:
  - Since a new instance is created on every request, objects created with `AddTransient` are eligible for garbage collection as soon as they go out of scope (when they are no longer referenced).
  - If these instances are lightweight and short-lived, they are typically collected in **Generation 0** of the GC, which is optimized for collecting short-lived objects.
- **Example**: `GuidGeneratorService` might generate a new GUID for every request. After the request is served and there are no more references to the service, the garbage collector will clean up these objects.

## 2. Scoped (`AddScoped`)

- **Behavior**: A single instance of the service is created and shared within the scope of an HTTP request (or any custom scope defined in the application). For each new HTTP request, a new instance is created.
- **Garbage Collection Context**:
  - Objects registered with `AddScoped` are longer-lived compared to `AddTransient` but are still short-lived. They will typically be collected after the HTTP request ends, as they go out of scope.
  - These objects will likely be collected in **Generation 1**, which is suitable for medium-lived objects.
- **Example**: `UserContextService` could maintain state (like user details or preferences) for the duration of an HTTP request. After the request is completed, the service instance will be eligible for garbage collection.

## 3. Singleton (`AddSingleton`)

- **Behavior**: A single instance of the service is created and shared across the entire application for its lifetime. The same instance is used throughout the application's runtime.
- **Garbage Collection Context**:
  - Objects registered with `AddSingleton` are typically long-lived. They are not eligible for garbage collection until the application is shut down or the DI container is disposed of.

    - Singleton services are usually moved to **Generation 2** of the garbage collector, which is reserved for long-lived objects. GC operations in Generation 2 are less frequent, as collecting long-lived objects is more costly in terms of performance.
- **Example**: `LoggingService` might manage log files or logging streams that should be available throughout the application's lifecycle. This makes it a good candidate for a singleton.

## Summary of GC Context for Each Lifetime

- **Transient**: Short-lived objects, quickly eligible for garbage collection (Gen 0).
- **Scoped**: Medium-lived objects, eligible for collection after the HTTP request ends (Gen 1).
- **Singleton**: Long-lived objects, live for the duration of the application (Gen 2).

—----------

## Conclusion

## Dispose Method

- When an object that implements the 'IDisposable' interface is no longer needed, the Dispose() method can be called to explicitly release unmanaged resources

- The Dispose() method calls the Dispose() method of any unmanaged objects that the object is using, to release their resources as well

- The Dispose() method then frees up any unmanaged resources that the object is using, such as file handles, terminating connections, or freeing up memory allocated outside of the managed heap

- Calling the Dispose() method ensures that the unmanaged resources used by the object are properly released, to avoid memory leaks and other issues

## Finalizers

Finalizers (historically referred to as destructors) are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector. In most cases, you can avoid writing a finalizer by using the System.Runtime.InteropServices.SafeHandle or derived classes to wrap any unmanaged handle.

- Finalizers cannot be defined in structs. They are only used with classes.
- A class can only have one finalizer.
- Finalizers cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.
- A finalizer does not take modifiers or have parameters.

For example, the following is a declaration of a finalizer for the Car class.

```
class Car
{
   ~Car()  // finalizer
   {
      // cleanup statements...
```

```
    }
}
```

The finalizer implicitly calls Finalize on the base class of the object. Therefore, a call to a finalizer is implicitly translated to the following code:

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

This design means that the Finalize method is called recursively for all instances in the inheritance chain, from the most-derived to the least-derived.

The programmer has no control over when the finalizer is called; the garbage collector decides when to call it. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for finalization, it calls the finalizer (if any) and reclaims the memory used to store the object.

**Using finalizers to release resources**

In general, C# does not require as much memory management on the part of the developer as languages that don't target a runtime with garbage collection. This is because the .NET garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources, such as windows, files, and network connections, you should use finalizers to free those resources. When the object is eligible for finalization, the garbage collector runs the Finalize method of the object.

## Using keyword

- The using keyword is used to create "using" directive, which specifies a namespace to be included in the program. This means all the types in that namespace can be used without having to qualify their names.

- For example, if a program needs to use types from the System.IO namespace, it can include a using directive for that namespace at the top of the source file like this:

Using System .IO

- The using keywords can also be used to create a "using" statement, which is used to automatically dispose of an object when it is no longer needed.

- This is useful for managing resources, such as file handles or database connections, that need to be closed when they are no longer needed

The `using` statement is syntactic sugar in C#. The C# compiler translates a `using` block into a `try-finally` block. This is what happens behind the scenes:

```
using (var fileStream = new FileStream("file.txt", FileMode.Open))
{
    // Use the fileStream
}
```

is translated to:

```
FileStream fileStream = new FileStream("file.txt", FileMode.Open);
try
{
    // Use the fileStream
}
finally
{
    if (fileStream != null)
    {
        ((IDisposable)fileStream).Dispose();
    }
}
```

## Stack overflow

- Occurs if call stack pointer exceeds the stack bound

- Call stack may consist of a limited amount of address space, often determined at the start of the program

- The most common cause of stack overflow is excessively deep recursion, in which a function calls itself so many times that the space needs to store the variables and information associated with each call more that can fit on the stack. An example of infinite recursion:

```
Public int foo()
{
    return foo();
}
```

- The function foo, when it is invoked, continues to invoke itself, allocating additional space on the stack each time, until the stack overflows resulting in segmentation fault.

## Stack memory allocation

- Stack allocation is the method of allocating memory on the stack in .NET. Some important features of stack allocation in .NET included:

- The stack is a region of memory that is used to store local variables and function arguments. It is organized as a last-in, first-out (LIFO) data structure, with the most recently allocated memory being the first to be deallocated.

- Stack allocation is faster than heap allocation, because the memory is automatically deallocated when a function or method completes execution. This means there is no need for a garbage collector to monitor the stack and reclaim memory.

- Stack allocation is generally used for short-lived variables, such as loop counters and function arguments, because the memory is automatically deallocated when the function or method completes execution.

- The size of the stack is limited, so it is important to avoid using excessive amounts of stack memory, as it may cause a stack overflow.

## Heap memory allocation

- Heap allocation is a method of allocating memory dynamically at runtime in .NET.

- Objects are stored in the heap, which is a region of memory that is managed by the .NET garbage collector

- The heap is divided into two parts: the small object heap and the large object heap. Small objects (less than 85,000 bytes) are stored in the small object heap, while large objects (greater than 85,000 bytes) are stored in the large object heap.

- Objects on the heap are accessed through references. When you create an object on the heap, you get a reference to the object on the stack.

- The garbage collector is responsible for managing the heap and reclaiming memory that is no longer being used. It runs automatically in the background and frees up memory by collecting and destroying objects that are no longer reachable.

- Heap allocation is slower than stack allocation because the garbage collector has to constantly monitor the heap for objects that are no longer being used and reclaim their memory.

## Reference types vs Value types

- Value types (derived from System.ValueType, e.g. int, bool, char, enum and struct) can be allocated on the stack or on the heap, depending on where they were declared

- If the value type was declared as a variable inside a method then it's stored on the stack

- If the value type was declared as a method parameter then it's stored on the stack

- If the value type was declared as a member of a class then it's stored on the heap, along with its parent

- A value type does hold the value to which it is associated. The example below shows a variable x, of type int(value type) and value 2. The block of memory associated with x therefore contains the integer 2 (i.e. its binary representation)

```
int x = 2;
```

- References are always stored on the stack with the thing that is being referenced stored on the heap

# 6a. Asynchronous Programming

## Threading

- Threading is a way for a program to run multiple tasks concurrently, or in parallel. In .NET, threading allows you to write programs that can take advantage of multiple processor cores on a computer, which can greatly improve performance of your program

- Threading is implemented using the System.Threading namespace in .NET, which contains classes and methods for creating and managing threads. For example, the System.Threading.Thread class provides methods for creating and starting new threads, as well as methods for managing the state of a thread

- To use threading in your .NET program, you can create a new thread and start it by calling the Thread.Start() method. You can then use the thread to run a task concurrently with the main thread of your program.

- Threading is useful for a wide variety of tasks, including parallelising computationally intensive operations, performing multiple tasks concurrently, and creating responsive user interfaces. It can also be used to improve the performance of your program by making better use of multiple processor cores on a computer

## Task

- .NET framework provides Threading.Tasks class which lets you create tasks and run them asynchronously. A task is an object that represents some work that should be done. The task can tell if the work is completed and if the operation returns a result, the task gives you a result.

## Difference between task and thread

- Task is abstraction on top of threads

- The thread class is used for creating and manipulating a thread in Windows. A task represents some asynchronous operation and is part of the Task Parallel library, a set of APIs for running tasks asynchronously and in parallel

- The task can return a result. There is no direct mechanism to return the result from a thread

- Task supports cancellation through use of cancellation tokens but thread doesn't. Can chain tasks

## Async await

- Used to designate methods that contain asynchronous code

- An example of how async and await might be used in .NET method:

```
public async Task<string> GetDataAsync()
```

```
{
    // Use the await keyword to wait for the task to complete.
    string data = await SomeAsyncMethod();
    // Now that the task is complete, you can use the data that was returned.
    return data;
}
```

- When async method is called, it will immediately return a Task object, allowing the calling code to continue executing without blocking.

- The await keyword is used to suspend execution of the GetDataAsync method until the SomeAsyncMethod task completes. Once the task completes, the await keyword will unblock the GetDataAsync method and allow it to continue executing.

## How Asynchronous Programming Works in .NET

**Initial Execution and Await**: When `GetDataAsync()` is called, it begins executing until it hits the `await` keyword:

```
string data = await SomeAsyncMethod();
```

1. At this point, the `await` keyword pauses the execution of `GetDataAsync()` until `SomeAsyncMethod()` completes. The thread that was executing `GetDataAsync()` (often a thread from the thread pool in ASP.NET) is released back to the thread pool to perform other work.
2. **Continuation After Await**: When `SomeAsyncMethod()` completes (such as when an I/O-bound operation finishes), the remaining logic in `GetDataAsync()` (after the `await`) needs to be resumed.
   By default, the continuation of the `GetDataAsync()` method will attempt to run on the same **SynchronizationContext** as it started with. In an ASP.NET context, this means the continuation will attempt to run on the original request thread that executed the method.
3. **What Happens if the Main Request Thread Is Busy?**
   - **ASP.NET Synchronization Context**: If the main request thread (the thread that initially called `GetDataAsync()`) is still busy doing other work, the continuation of `GetDataAsync()` will **not block** until that thread is free. Instead, the continuation will be queued to run on the request's synchronization context. Once the original request thread is free, the continuation will execute. If the request thread is handling something that prevents it from being freed up in a timely manner, there could be a delay.
   - **Thread Pool Threads**: In ASP.NET Core, the default `SynchronizationContext` is not present. Instead, the continuation will execute on any available thread from the thread pool, not necessarily the original request thread. This allows for more efficient handling since it avoids the restriction of using the original thread, and can resume on any free thread, improving scalability.

## What Happens When the Main Request Thread Is Busy?

- **ASP.NET (Classic)**: If the main request thread is busy when the asynchronous operation completes, the continuation will wait for that thread to be free. This could potentially cause a delay in executing the continuation, especially if the request thread is blocked or doing heavy work.

- **ASP.NET Core**: If you use `.ConfigureAwait(false)` after the `await`, the continuation is not dependent on the original context and can resume on any available thread from the thread pool. This is a more scalable and efficient model as it avoids thread starvation and delays.

## Using `.ConfigureAwait(false)`

To avoid being bound to the original request thread (and to potentially improve performance and scalability), you can use `.ConfigureAwait(false)`:

```
public async Task<string> GetDataAsync()
{
    // Use the await keyword to wait for the task to complete.
    string data = await SomeAsyncMethod().ConfigureAwait(false);
    // Now that the task is complete, you can use the data that was returned.
    return data;
}
```

- By using `.ConfigureAwait(false)`, you indicate that the continuation of `GetDataAsync()` does not need to run on the original request thread. This allows .NET to use any available thread from the thread pool, reducing the risk of blocking or delaying due to a busy request thread.

## Conclusion

- **ASP.NET**: The continuation might be delayed if the main request thread is still busy since it will try to run on the original context.
- **ASP.NET Core**: Without a `SynchronizationContext`, the continuation can resume on any available thread, avoiding delays.
- Using `.ConfigureAwait(false)` ensures that the continuation does not depend on the original thread, promoting better scalability and responsiveness.

**Task.Run** - Use Task.Run to offload CPU-bound work to a background thread.

```
using System;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        Console.WriteLine("Starting CPU-bound work...");

        // Offload the CPU-bound operation to a background thread
        int result = await Task.Run(() => CalculateFactorial(20));

        Console.WriteLine($"Factorial result: {result}");
        Console.WriteLine("CPU-bound work completed.");
    }

    // A CPU-bound method that calculates the factorial of a number
    private static int CalculateFactorial(int number)
    {
        int result = 1;
        for (int i = 1; i <= number; i++)
        {
            result *= i;
        }

        Console.WriteLine($"Factorial calculation done on Thread
{Thread.CurrentThread.ManagedThreadId}");
        return result;
    }
```

Parallel programming and asynchronous programming are two different approaches to managing tasks, particularly in applications that require concurrency or need to handle multiple operations simultaneously. Here's a breakdown of the key differences between them:

## Parallel Programming

1. **Definition:**
    - Parallel programming is a technique used to perform multiple tasks or computations simultaneously. It involves dividing a problem into smaller parts and executing these parts in parallel across multiple processors or cores.
2. **Purpose:**
    - The primary goal of parallel programming is to improve the performance of computationally intensive tasks by utilizing multiple CPU cores to perform computations faster.
3. **How it Works:**
    - In parallel programming, multiple threads or tasks run at the same time on different CPU cores. For example, if a task can be divided into four independent subtasks, parallel programming will execute these subtasks on four separate threads or cores simultaneously.
    - Libraries such as `Parallel.For`, `Parallel.ForEach`, and `Task Parallel Library (TPL)` in .NET can be used to perform parallel operations.
4. **Use Case:**
    - **CPU-bound tasks** that require heavy computation, such as mathematical calculations, image processing, machine learning model training, etc.
5. **Example in .NET:**

```
Parallel.For(0, 10, i =>
{
    // Each iteration runs in parallel on separate threads
    Console.WriteLine($"Processing iteration {i} on Thread
{Thread.CurrentThread.ManagedThreadId}");
});
```

6. **Concurrency:**
    - Achieves concurrency by dividing a task into smaller sub-tasks that run simultaneously across multiple threads.
7. **System Resources:**
    - Uses multiple CPU cores to increase the performance of computations.

## Asynchronous Programming

1. **Definition:**
    - Asynchronous programming is a technique used to perform tasks without blocking the execution thread. The main thread can continue doing other work while waiting for a time-consuming operation (such as I/O-bound tasks) to complete.
2. **Purpose:**
    - The primary goal of asynchronous programming is to improve the responsiveness of an application, especially when dealing with tasks that involve waiting, such as I/O operations (reading files, making HTTP requests, etc.).
3. **How it Works:**

- In asynchronous programming, an `async` task is started and the main thread is freed up to handle other work. The `await` keyword is used to specify that the main thread should continue only after the asynchronous task completes.
- The system uses a single thread (or a small pool of threads) to perform multiple tasks by switching context, but not necessarily at the same time.

4. **Use Case:**
   - **I/O-bound tasks** that involve waiting for resources, such as network requests, database calls, file I/O operations, etc.
5. **Example in .NET:**

```
public async Task FetchDataAsync()
{
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync("https://example.com");
    Console.WriteLine("Data fetched successfully.");
}
```

6. **Concurrency:**
   - Achieves concurrency by overlapping tasks, using the same thread to start an operation, wait, and continue without blocking.
7. **System Resources:**
   - Does not necessarily use multiple CPU cores; instead, it makes better use of I/O operations to avoid blocking.

## Key Differences:

| Aspect | Parallel Programming | Asynchronous Programming |
|---|---|---|
| **Purpose** | Speed up CPU-bound tasks | Improve responsiveness for I/O-bound tasks |
| **Concurrency** | Multiple tasks run simultaneously on different threads or cores | Multiple tasks wait for I/O or other operations without blocking the main thread |
| **Resources Used** | Multiple CPU cores | Minimal resources, single or few threads |

| | | |
|---|---|---|
| **Use Case** | Computationally intensive tasks | I/O-bound tasks (file I/O, network calls) |
| **Execution** | Tasks are executed in parallel | Tasks are executed asynchronously and may be interleaved |
| **Blocking Behavior** | May block if all threads are occupied | Non-blocking, frees up threads while waiting |
| **Example Scenario** | Running matrix calculations | Fetching data from an API |

### Conclusion:

- **Parallel programming** is suitable for CPU-intensive operations where tasks can be split and executed simultaneously.
- **Asynchronous programming** is best for I/O-bound operations where tasks involve waiting for external resources, such as network requests or database calls.

Choosing between the two approaches depends on the type of task you are dealing with—whether it's CPU-bound or I/O-bound—and your specific requirements for performance and responsiveness.

# 6b. Parallel Programming

- **Parallel.ForEach:**
  - Use Parallel.ForEach to execute iterations of a loop in parallel, which can improve performance for CPU-bound tasks.
- **PLINQ (Parallel LINQ):**
  - Use AsParallel() to enable parallel execution of LINQ queries. This is useful for large datasets where operations can be executed concurrently.
- **Task and CancellationToken:**
  - Use CancellationToken to support canceling tasks gracefully.
  - Example:

```
var cts = new CancellationTokenSource();
var token = cts.Token;
Task.Run(() =>
{
    // Long-running operation
    for (int i = 0; i < 1000; i++)
    {
        if (token.IsCancellationRequested)
            break; // Cancel the operation
    }
}, token);
```

**Best Practices:**

- Avoid parallelizing I/O-bound tasks as it may lead to thread exhaustion.
- Use Task.WhenAll or Task.WhenAny to manage multiple tasks concurrently.

# 7. LINQ

Here are your notes in a format you can easily copy into a Word document:

---

**Common LINQ Operators**

## Filtering Operators

**Where:** Filters elements based on a predicate.

```
var adults = people.Where(p => p.Age >= 18);
```

## Projection Operators

**Select:** Projects each element into a new form.

```
var names = people.Select(p => p.Name);
```

**SelectMany:** Projects each element into a collection and **flattens** the resulting collections into a single sequence.

```
var allCourses = students.SelectMany(s => s.Courses);
```

## Sorting Operators

**OrderBy / OrderByDescending:** Sorts elements in ascending or descending order.

```
var sortedByAge = people.OrderBy(p => p.Age);
```

**ThenBy / ThenByDescending:** Provides a secondary sort.

```
var sortedByAgeAndName = people.OrderBy(p => p.Age).ThenBy(p =>
p.Name);
```

## Grouping Operators

**GroupBy:** Groups elements by a specified key.

```
var groupedByCity = people.GroupBy(p => p.City);
```

## Quantifiers

**Any:** Determines if any elements satisfy a condition.

```
bool hasAdults = people.Any(p => p.Age >= 18);
```

**All:** Determines if all elements satisfy a condition.

```
bool allAdults = people.All(p => p.Age >= 18);
```

## Aggregation Operators

**Count:** Counts the number of elements.

```
int count = people.Count();
```

**Sum / Min / Max / Average:** Performs aggregation on numeric data.

```
int totalAge = people.Sum(p => p.Age);
```

## Set Operators

**Distinct:** Removes duplicate elements.

```
var uniqueCities = people.Select(p => p.City).Distinct();
```

**Union / Intersect / Except:** Set operations for combining sequences.

```
var allCities = citiesInUs.Union(citiesInCanada);
```

## Element Operators

**First / FirstOrDefault:** Returns the first element (or the first matching element) or a default value if no elements are found.

```
var firstPerson = people.FirstOrDefault(p => p.Age >= 18);
```

**Single / SingleOrDefault:** Returns the only element or throws an exception if more than one element is found. SingleOrDefault returns the default value if no elements are found.

```
var thePerson = people.Single(p => p.Id == 1);
```

## Join Operators

**Join:** Joins two sequences based on matching keys.

```
var result = people.Join(cities, p => p.CityId, c => c.Id, (p, c) =>
new { p.Name, c.Name });
```

**GroupJoin:** Groups the results of a join.

```
var result = cities.GroupJoin(
    people,
    c => c.Id,
    p => p.CityId,
    (c, peopleGroup) => new { City = c.Name, People =
peopleGroup.Select(p => p.Name) }
);
```

## Understanding Deferred Execution in LINQ

**Deferred execution** in LINQ means that a query is not executed when it is defined but rather when it is iterated over (e.g., using a `foreach` loop or converting it to a collection like a `List`).

This feature allows LINQ to provide efficient query execution and work with the most up-to-date data.

### Example of Deferred Execution

Consider the following example:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

var evenNumbers = numbers.Where(n => n % 2 == 0);

// Deferred execution: The 'Where' query is not executed yet
numbers.Add(6);

foreach (var num in evenNumbers)  // Now the query is executed
{
    Console.WriteLine(num);  // Output: 2, 4, 6
}
```

- The `Where` LINQ method does not immediately filter the `numbers` list.
- When we add `6` to `numbers`, and **only when we iterate** over `evenNumbers` with `foreach`, the LINQ query is executed and includes `6` in the results.

### Why Call `ToList()`?

If you **want to avoid deferred execution** and **materialize the results immediately**, you can use `ToList()`:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

var evenNumbers = numbers.Where(n => n % 2 == 0).ToList();  //
Immediate execution

numbers.Add(6);

foreach (var num in evenNumbers)
{
    Console.WriteLine(num);  // Output: 2, 4
}
```

- Here, `ToList()` forces the `Where` query to execute immediately, creating a new `List<int>` containing the current even numbers.

- Adding `6` to `numbers` after calling `ToList()` does not affect `evenNumbers` because the query was already executed, and the results were stored in a separate list.

### PLINQ

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        // Create a large list of numbers
        List<int> numbers = Enumerable.Range(1, 1000000).ToList();

        // Perform parallel execution of the query
        var squaredNumbers = numbers.AsParallel().Select(number =>
        {
            // Simulate a time-consuming computation
            int square = number * number;
            Console.WriteLine($"Square of {number} is {square} (Processed by
Thread {System.Threading.Thread.CurrentThread.ManagedThreadId})");
            return square;
        }).ToList();

        Console.WriteLine("All tasks are complete.");
    }
}
```

# 7. .NET Framework and .NET Core

## .NET overview

.NET is a free, cross-platform, open-source developer platform for building many different types of applications. It consists of several key components:

**Runtime Environments:**

.NET Core: A cross-platform runtime for cloud, IoT, and desktop apps.
.NET Framework: The original runtime for Windows desktop applications and web services.
Mono: An open-source implementation of the .NET Framework, primarily used for running .NET applications on macOS and Linux.
Languages: .NET supports multiple programming languages, including C#, F#, and Visual Basic.

**Class Libraries:** These are pre-built code libraries that provide a wide array of functionalities, such as file input/output, database interaction, web application development, and more.

**Base Class Library (BCL):** A core set of libraries that are part of the .NET standard, providing common functionality across all .NET environments.

**Common Language Runtime (CLR):** A runtime environment that manages the execution of .NET programs, providing services like memory management, type safety, exception handling, garbage collection, and more.

**Frameworks and Tools:**

ASP.NET: For building web applications.
Entity Framework: For database operations.
WinForms and WPF: For desktop applications.
Xamarin: For mobile application development.
.NET Standard: A formal specification of .NET APIs that are intended to be available on all .NET implementations.

In summary, .NET is a platform that provides a comprehensive environment for building a wide range of applications using various languages (including C#), libraries, and tools. C# is a programming language that is designed to work seamlessly with the .NET platform, though it can also be used in other contexts. C# is one of the primary languages used for writing applications on the .NET platform.

## Code execution process

The code execution process involves the following two stages:

1. Compiler time process
2. Runtime process

- When the compiler compiles, C# code must get converted into machine code. However it is first converted into intermediate language code. IL code is partially compilled code
- IL code is partially compiled code. We have the JIT compiler which runs over the IL and compiles it into machine language

## Common Type System (CTS)

- CTS ensures that data types defined in two different languages get compiled into a common data type

## Common Language System (CLS)

- Set of rules and guidelines for designing libraries and components that are intended to be used by multiple porgramming languages.

- It is a subset of the .NET framework that defines a set of requirements for language interoperability

- The CLS is designed to ensure that libraries and components written in one .NET language can be used by other .NET languages

- For example, if a library is written in C# and follows the CLS, it can be used by a program written in Visual Basic or any other .NET languages

- The CLS defines a set of rules for naming conventions, data types, and other aspects of library design that are intended to ensure that libraries are easy to use and understand for programmers working in any .NET language.

### Managed vs Unmanaged code

- Code that executes under the CLR environment is managed code. The CLR converts it to native language, the garbage collector frees up memory when objects are not being used.

- C++ is unmanaged code. They have their own compilers, their own runtimes that CLR doesn't control.

- Managed code is code that runs under the control of the CLR execution environment and unmanaged code is code that runs outside the control of the CLR. This code has its own runtime, compiler etc. Own way of doing memory allocation.

### Assemblies

An assembly is a self-contained unit of functionality that is developed and deployed as a single entity. It is the basic unit of deployment in .NET, and can be an executable (.exe) or a library (.dll). Assemblies contained compiled code (usually in the form of a Microsoft Intermediate Language (MSIL)).

### DLL file

In the .NET framework, a DLL (Dynamic Link Library) is a type of file that contains compiled code that can be used by other programs. DLL files can contain a variety of different types of code, including class libraries, utility functions, and graphical user interface elements.

# 8. Web Development

### ASP.NET and ASP.NET Core: Building Web Applications

#### ASP.NET Overview

ASP.NET is a mature web framework, part of the .NET Framework, for building web applications using .NET. It supports the development of dynamic web pages, web services, and web APIs.

#### Key Features

**Web Forms:** For creating dynamic web applications using a rich component-based model.

**MVC Pattern:** ASP.NET MVC offers a powerful, patterns-based way to build dynamic websites enabling a clean separation of concerns.

**Web API:** Building RESTful services that can be consumed by a variety of clients including browsers and mobile devices.

#### ASP.NET Core

ASP.NET Core is a redesign **of** ASP.NET, cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications.

**Advancements:** Offers improved performance, reduced footprint, modular components, and supports cross-platform development.

**Features:** Dependency injection, asynchronous programming models, unified story for building web UI and APIs.

**Razor Pages:** A simpler way to organize code for pages-based scenarios.

**Blazor:** Enables running C# in the browser on WebAssembly, creating interactive web UIs with C# instead of JavaScript.

## RESTful Services:

**Concept:** RESTful web services implement REST (Representational State Transfer) architecture, a lightweight and maintainable approach for building web services.

**ASP.NET Web API:** Ideal for creating RESTful applications on the .NET Framework.

**ASP.NET Core:** Offers improved tools for building RESTful services, including integration with popular open-source projects.
Best Practices: Stateless design, use of HTTP methods (GET, POST, PUT, DELETE), and HTTP status codes.

tatehe `HttpClient` class in .NET is a powerful and flexible class used to send HTTP requests and receive HTTP responses from a resource identified by a URI (Uniform Resource Identifier). It's commonly used for consuming web APIs, fetching resources from the web, or making HTTP requests to external services in an asynchronous manner.

## Overview of the `HttpClient` Class

### Key Characteristics:

1. **Asynchronous Operations**: The `HttpClient` class is designed with async/await support, making it easy to perform non-blocking HTTP operations.
2. **Supports Various HTTP Methods**: It supports all standard HTTP methods such as `GET`, `POST`, `PUT`, `DELETE`, `PATCH`, `HEAD`, and `OPTIONS`.
3. **Reusable and Configurable**: `HttpClient` can be reused for multiple requests, which allows efficient management of connections, especially when communicating frequently with the same server.
4. **Supports Content Negotiation**: It can handle different content types, including JSON, XML, plain text, etc.
5. **Extensibility**: Allows customization of request headers, timeout settings, authentication, and more.

### Basic Usage:

To use `HttpClient`, you typically need to:

1. **Instantiate an `HttpClient` object**.
2. **Configure the `HttpClient`** (optional).
3. **Send HTTP requests** using methods like `GetAsync`, `PostAsync`, `PutAsync`, etc.
4. **Handle HTTP responses** by reading the content, checking status codes, and managing errors.

## Example Usage:

Below is a simple example demonstrating how to use `HttpClient` to make a GET request to a public API.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        // Create a single instance of HttpClient
        using (HttpClient httpClient = new HttpClient())
        {
            try
            {
                // Set a base address for all requests
                httpClient.BaseAddress = new
Uri("https://jsonplaceholder.typicode.com/");

                // Send a GET request
                HttpResponseMessage response = await
httpClient.GetAsync("posts/1");

                // Check if the request was successful
                if (response.IsSuccessStatusCode)
                {
                    // Read the response content as a string
                    string content = await response.Content.ReadAsStringAsync();
                    Console.WriteLine("Response content:");
                    Console.WriteLine(content);
                }
                else
                {
                    Console.WriteLine($"Error: {response.StatusCode}");
                }
            }
            catch (HttpRequestException ex)
            {
                Console.WriteLine($"Request error: {ex.Message}");
            }
        }
    }
```

```
}
```

### Explanation of the Example:

1. **Instantiating `HttpClient`**:
   - A single instance of `HttpClient` is created inside a `using` statement to ensure proper disposal after use.
2. **Setting the Base Address**:
   - `BaseAddress` is set to the root URL of the API, simplifying requests by only needing to provide relative paths.
3. **Sending a GET Request**:
   - `GetAsync` method sends a GET request asynchronously to the specified endpoint (`"posts/1"`).
4. **Handling the Response**:
   - The response is checked for success using `IsSuccessStatusCode`.
   - The content is read asynchronously using `ReadAsStringAsync` if the request was successful.
5. **Error Handling**:
   - A `try-catch` block catches `HttpRequestException` for any issues during the request.

## Key Properties and Methods:

- **Properties:**
  - `BaseAddress`: The base address of the Uniform Resource Identifier (URI) of the Internet resource used when sending requests.
  - `DefaultRequestHeaders`: The default headers included in every request sent using this `HttpClient`.
  - `Timeout`: The timespan to wait before the request times out.
- **Methods:**
  - `GetAsync(string requestUri)`: Sends a GET request to the specified URI.
  - `PostAsync(string requestUri, HttpContent content)`: Sends a POST request with the specified content to the URI.
  - `PutAsync(string requestUri, HttpContent content)`: Sends a PUT request with the specified content.
  - `DeleteAsync(string requestUri)`: Sends a DELETE request.
  - `SendAsync(HttpRequestMessage request)`: Sends an HTTP request as an asynchronous operation.
  - `GetStringAsync(string requestUri)`: Sends a GET request and returns the response body as a string.

## Best Practices:

1. **Reusing `HttpClient` Instances**:
   - `HttpClient` should be reused throughout the application lifecycle to avoid exhausting the number of available sockets.
   - You can use dependency injection or a singleton pattern to manage `HttpClient` instances.

2. **Setting Timeouts**:
    - Set reasonable timeout values to prevent requests from hanging indefinitely.
3. **Handling Exceptions**:
    - Be prepared to handle various exceptions such as `HttpRequestException` for HTTP errors and `TaskCanceledException` for timeouts.
4. **Avoid Using `HttpClient` in `using` Statements**:
    - Instantiating `HttpClient` within a `using` block can lead to socket exhaustion issues because the underlying connections are not properly released.

### Common Use Cases:

- **Web API Consumption**: Interacting with RESTful APIs to send and receive data.
- **Web Scraping**: Fetching HTML content from websites.
- **Microservices Communication**: Communicating between microservices in a distributed system.

`HttpClient` is a versatile and essential class in .NET for making HTTP requests and is widely used in both client and server-side applications.

# 9. Database Access

### Connecting to a DataSource in Ado .NET

In Ado .NET, you use a connection object to connect to a specific data source by supplying necessary authentication information in a connection string. The Connection Object you use depends on the type of data source.

### SQL join

Used to combine rows from two or more tables based on a related column between them

### Inner join

Selects record that have matching values in both tables

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

Table1 is user and table2 is posts

### Left (Outer) join

Returns all records from the left table, and the matched records from the right table

```
SELECT column_name (s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

table1 is user and table2 is posts

## Right (Outer) join

Returns all records from the right table, and then matching records from the left table

```
SELECT column_name (s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

table1 is user and table2 is posts

## Full (Outer) join

Returns all records when there is a match in either left or right table

```
SELECT column_name (s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```

Table1 is user and table2 is posts

## Join table / multiple joins

Given the following tables:

**Image**

Use the following query:

```
SELECT *
    FROM RECIPES r
    JOIN RECIPES ri ON ri.recipe_id = r.id
    JOIN INGREDIENTS i ON i .id = ri.ingredients_id AND i.name IN ('chocolate', 'cream')
```

## Difference between SQL and NoSQL database

Five critical differences between SQL and NoSQL databases:

1. SQL databases are relational. NoSQL databases are non-relational
2. SQL databases are structured query language and have a predefined schema. NoSQL databases have dynamic schemas for unstructured data
3. SQL databases are vertically scalable, while NoSQL databases are horizontally scalable
4. SQL databases are table-based, while NoSQL databases are document, key-value, graph, or wide-column stores
5. SQL databases are better for multi-row transactions, while NoSQL is better for unstructured data like documents or JSON

## Database Indexing

- Technique used to improve performance of database queries.

- An index is a data structure that allows efficient retrieval of data from a database table

- When a database table is indexed, a separate data structure is created that contains a reference to each record in the table. This data structure is organised in such a way to allows fast searching and sorting of records in the table

- For example, imagine you have a table of customer records, with each record containing information about a particular customer, such as their name, address, and phone number. If you want to find a particular customer by their name, you would have to search through every record in the table to find the one you're looking for.

- However, if this table is indexed by name, the database can use the index to quickly find the record you're looking for, without having to search through every record in the table. This can greatly improve the performance of your database queries, especially on large tables

In database systems, indexes are used to speed up the retrieval of data from a table. There are two main types of indexes: clustered and non-clustered, and they differ in how they store data and impact data retrieval.

**Clustered Index:**

**Storage:** In a clustered index, the data in the table is physically stored in the order of the index. There can be only one clustered index per table, as the data rows themselves can be sorted in only one order.
**Data Retrieval:** Since the data is stored in index order, reading data using the clustered index can be very fast, especially for range queries.
**Modification Impact:** Inserting, updating, or deleting rows can be more expensive, as the data rows might need to be moved to maintain order.
**Usage:** It's typically used for columns that are often accessed in a sequential manner, such as primary keys.

**Non-Clustered Index:**

**Storage:** A non-clustered index creates a separate structure from the data rows. It contains a copy of the indexed column’s data along with a pointer to the data row that contains the corresponding value. As such, multiple non-clustered indexes can be created on a table.
**Data Retrieval:** When a non-clustered index is used, the database first looks up the index to find the location of the data row, then retrieves the data from the table. This two-step process can be slower than using a clustered index for the same data.
**Modification Impact:** Modifications are generally faster than with a clustered index, as the data rows themselves don't need to be moved; only the index needs to be updated.
**Usage:** Non-clustered indexes are suitable for columns used in search conditions (WHERE clauses) or for indexing a wide range of columns.

In summary, the choice between clustered and non-clustered indexes depends on the nature of the data, the types of queries run against the database, and the specific performance requirements. A clustered index is ideal for columns that are frequently accessed sequentially, while non-clustered indexes are better for columns used in various search conditions and where multiple indexes on different columns are needed.

# 10. Software Design and Architecture

## Four pillars of OOP

### Encapsulation

- Each object in your code should control its own state
- Put implementation into private methods so that these methods cannot be called from outside the object

### Inheritance

- Lets one object acquire the properties and methods of another object
- Related in the Liskov substitution principle. It states that if you can use a parent class anywhere you use a child - and ChildType inherits from ParentType - then you pass the test
- Use shape example

### Polymorphism

- Compile-time Polymorphism (Method Overloading)
- Run-time Polymorphism (Method Overriding)

### Abstraction

- Finding things that are generic in your code and providing a generic function or object to serve multiple places/with multiple concerns

## SOLID Principles

### Single Responsibility Principle

- A class should have one and only one reason to change, meaning that a class should have only one job

### Open-Closed Principle

- A class should be extendable without modifying the class itself

### Liskov Substitution Principle

- This means every subclass or derived class should be substitutable for their base or parent class
- Subtypes should not break the contracts set by their parent types. In practical terms, the virtual and override methods are substitutable.

### Interface Segregation Principle

- A client should never be forced to implement an interface that it does not use, or clients shouldn't be forced to depend on methods they do not use.
- It is better to have many small, specific interfaces than a few large, general ones.
- Used to improve the design of a program by making it more modular and adaptable

### Dependency Inversion Principle

- States that high-level modules should not depend on low-level modules, but rather should depend on abstractions. Ie. the design of a program should not depend on the details of how its components are implemented but rather on the interfaces that define how those components interact
- E.g. if a high-level module in your program uses a database class, it should be implemented behind an abstraction such as an interface called IDatabase

## Clean code

- Readable
- Consistent
- Maintainable
- Documented

## Model View Presenter design pattern

- A derivation of the model-view-controller (MVC) architectural pattern, and it is used mostly for building interfaces

- The pattern is used to separate the logic of the application from its user interface, allowing for more modular and testable code

- In MVP, the presenter acts as a middleman between the model and the view. The model represents the data and business logic of the application, while the view is the user interface that displays data to the user

- The presenter is responsible for updating the view with data from the model and handling any user interactions with the view. This separation of concerns makes it easier to develop, test, and maintain the application.

- One of the key benefits of using the MVP pattern is that it allows for clear separation of concerns between different components of the application

- This makes it easier to modify and update the application without affecting other parts of the code. Additionally, because the presenter is responsible for updating the view, the view can be easily replaced without affecting the underlying business logic of the application. This can be useful when developing applications for multiple platforms or when making significant changes to the user interface

## Model View Controller design pattern

- Commonly used in development of user interfaces. Pattern is used to separate logic of the application from its user interface, allowing for more modular and testable code.

- In MVC, the controller acts as a middleman between the model and the view. The model represents the data and business logic of the application, while the view is the user interface that displays the data to the user.

- The controller is responsible for handling user input and updating the view and the model as necessary. This separation of concerns makes it easier to develop, test, and maintain the application.

- One of the key benefits of using the MVC pattern is that it allows for a clear separation of concerns between different components of the application.

- This makes it easier to modify and update the application without affecting the other parts of the code.

- Additionally, because the controller is responsible for handling user input and updating the view and the model, the view and the model can be easily replaced without affecting the underlying business logic of the application.

- This can be useful when developing applications for multiple platforms or when making significant changes to the user interface

# 11. Testing and Debugging

## Unit Testing

Unit testing involves testing individual components of an application in isolation to ensure that each part functions correctly.

**Frameworks:**

**NUnit:** One of the earliest testing frameworks for .NET, inspired by JUnit. It provides a wide range of assertions and is known for its simplicity and effectiveness.

**xUnit:** A more modern framework, xUnit is often praised for its extensibility and support for parallel test execution.

**MSTest:** Microsoft's official testing framework, integrated with Visual Studio. It's convenient for developers working within the Microsoft ecosystem.

**Evolution:** Over time, these frameworks have incorporated features like mocking, data-driven testing, and integration with CI/CD pipelines.

## Integration Testing

Focuses on testing the interactions between different parts of the application, as well as the application's interactions with external systems like databases or web services.

**Strategies**

**Test Environments:** Setting up environments that mimic production to ensure tests are realistic.

**Mocking and Stubbing:** Using tools like Moq or NSubstitute to simulate external dependencies.

**Continuous Integration (CI):** Automated integration testing as part of CI pipelines to catch issues early in the development cycle.

**Trends:** Integration testing has become more automated and integrated into the software development lifecycle, particularly with the rise of DevOps and Agile methodologies.

# 12. Using .NET with Docker and Kubernetes

**Docker** and **Kubernetes** are two essential technologies for modern application development and deployment. They are often used together to containerize applications and orchestrate their deployment, scaling, and management in production environments.

### 1. Docker: Containerization of .NET Applications

**Docker** is a platform that allows you to package applications and their dependencies into containers. A container is a lightweight, portable, and self-sufficient unit that contains everything needed to run an application, including the runtime, libraries, and configuration files.

- **Containerization Benefits**:
  - **Isolation**: Each .NET application runs in its own isolated environment, preventing conflicts between dependencies.
  - **Portability**: Containers can run consistently across different environments (development, testing, production).
  - **Efficiency**: Containers are lightweight and start quickly compared to virtual machines, reducing resource usage and costs.

#### Steps to Containerize a .NET Application Using Docker:

1. **Create a Dockerfile**: A Dockerfile is a text file that contains instructions for building a Docker image of your .NET application.
   Example Dockerfile for a .NET application:

```
    # Use the official .NET SDK image to build the application
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /app

# Copy the project files and restore dependencies
COPY *.csproj .
RUN dotnet restore

# Copy all source files and build the application
COPY . .
RUN dotnet publish -c Release -o out

# Use the official ASP.NET runtime image to run the application
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS runtime
WORKDIR /app
COPY --from=build /app/out .
ENTRYPOINT ["dotnet", "YourApp.dll"]
```

2. **Build the Docker Image**:

Run the following command in the directory containing the Dockerfile:
bash
Copy code
```
docker build -t yourapp:latest .
```

3. **Run the Docker Container**:

Start a container from the built image:

```
docker run -d -p 8080:80 yourapp:latest
```

### 2. Kubernetes: Orchestration and Management of Containers

**Kubernetes** is an open-source platform designed to automate the deployment, scaling, and management of containerized applications.

- **Key Features of Kubernetes**:
  - **Orchestration**: Manages the deployment of multiple containers across a cluster of machines.
  - **Scaling**: Automatically scales applications up or down based on traffic or load.
  - **Self-Healing**: Automatically restarts failed containers and reschedules them across nodes if necessary.
  - **Load Balancing**: Distributes network traffic across multiple instances of an application to ensure reliability and performance.

### Steps to Deploy a .NET Application to Kubernetes:

1. **Create a Kubernetes Deployment File**: This YAML file defines how the application should be deployed and managed in a Kubernetes cluster.
   Example `deployment.yaml` for a .NET application:

```
    apiVersion: apps/v1
kind: Deployment
metadata:
  name: yourapp-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: yourapp
  template:
    metadata:
      labels:
        app: yourapp
    spec:
      containers:
      - name: yourapp
        image: yourapp:latest
```

```
        ports:
        - containerPort: 80
```

**2. Deploy to Kubernetes**:

Apply the deployment file to the Kubernetes cluster:

```
kubectl apply -f deployment.yaml
```

3. **Expose the Application**:

Use a Kubernetes Service to expose the application outside the cluster:

```
apiVersion: v1
kind: Service
metadata:
  name: yourapp-service
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: yourapp
```

Apply the service file:

```
kubectl apply -f service.yaml
```

### 3. Using Docker and Kubernetes Together

- **Docker** is used for creating and packaging your .NET applications into containers, ensuring consistency and portability.
- **Kubernetes** is used to deploy, scale, and manage these containers across multiple nodes in a cluster, providing high availability, load balancing, and automated management.

## Benefits of Using Docker and Kubernetes for .NET Applications:

- **Simplified Deployment**: Easily deploy your .NET applications across different environments.
- **Scalability**: Automatically scale your applications based on demand.
- **Resilience and High Availability**: Kubernetes' self-healing capabilities ensure that your applications are always running optimally.
- **Resource Efficiency**: Containers use fewer resources than traditional virtual machines, allowing for more efficient utilization of hardware.

## Conclusion:

Using Docker with Kubernetes provides a powerful solution for developing, deploying, and managing .NET applications in modern cloud environments. It enables consistent deployment, automated scaling, and efficient resource utilization, making it ideal for both small applications and large, distributed systems.

# 13. Other

### “==” vs “===” in Javascript

- The "==" operator performs a "loose" comparison which means that it will compare the values of the operands after converting them to a common type

- The "===" opeartor performs a "strict" comparison, which means that it will compare the values of the operands  without converting them to a common type.

```
// Comparison of a string and a number with ==
var m = "1";
var n = 1;
console.log(m == n); // Output: true

// Comparison of a string and a number with ===
var t = "1";
var u = 1;
console.log(t === u); // Output: false
```