# Define C#

* C# (pronounced "C sharp") is a modern, versatile, and object-oriented programming language, that is case sensitive, developed by Microsoft. It is Microsoft's flagship .NET Framework language which bears similarities to the C++ and Java languages and is commonly used for building various types of applications, including desktop, web, mobile, and cloud-based applications.

# Define .NET, .Net Framework, Asp.Net MVC. .Net Core, .Net Standard

* In summary, .NET is a comprehensive platform, the .NET Framework is the original implementation primarily for Windows applications, ASP.NET MVC is a web application framework within ASP.NET, .NET Core is a cross-platform framework for modern applications, and .NET Standard is a specification for creating cross-platform libraries. With the release of .NET 5 and later, Microsoft has unified the different variants into a single platform, simply referred to as .NET.

## .NET:

* .NET is a free, open-source, cross-platform framework developed by Microsoft. It provides a runtime environment (Common Language Runtime - CLR) and a comprehensive set of libraries and frameworks for building different types of applications, including desktop, web, mobile, and cloud applications.

## .NET Framework:

* The .NET Framework is the original implementation of the .NET platform. It is primarily designed for Windows applications and includes a large set of libraries and APIs. While it is widely used, .NET Framework is now considered legacy, and Microsoft's focus has shifted to newer technologies like .NET Core and .NET 5 and later.

## ASP.NET MVC (Model-View-Controller):

* ASP.NET MVC is a web application framework within the ASP.NET framework that implements the Model-View-Controller architectural pattern. It provides a structured way to build dynamic, scalable, and maintainable web applications. ASP.NET MVC separates the application into three main components: Model (data and business logic), View (user interface), and Controller (handles user input and updates the model and view).

## .NET Core:

* .NET Core is an open-source, cross-platform, and modular framework designed for developing modern, cloud-based, and cross-platform applications. It is a successor to the .NET Framework and is optimized for performance, scalability, and flexibility. .NET Core allows developers to build applications that can run on Windows, macOS, and Linux.

## .NET Standard:

* .NET Standard is a specification that establishes a set of APIs that must be available on all .NET implementations. It is a way to create libraries that can be shared across different .NET platforms, including .NET Framework, .NET Core, and Xamarin. .NET Standard simplifies the process of creating cross-platform libraries by defining a common set of APIs that are supported on various .NET platforms.

# What is OOP?

* OOP or Object-Oriented Programming is widely used for designing and building complex software systems.
* It is an approach to software design that revolves around the concept of "objects." Objects are instances of classes, which act as blueprints defining both data and actions associated with real-world entities.
* OOP emphasizes organizing code based on the properties and behaviors of these objects, promoting encapsulation, inheritance, and polymorphism to enhance code modularity, reusability, and maintainability.
* It's a design philosophy that mirrors the dynamic interactions of the real world within a codebase. It's a way of structuring and designing code based on real-world entities and their interactions.
* Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of "objects." An object is a self-contained unit that encapsulates data and behavior, interacting with other objects in a well-defined manner. OOP is based on four main principles, known as the four pillars of OOP:

# Four Pillars of OOP

## Encapsulation

* + A screenshot of a computer code

    Description automatically generatedis like a treasure chest. It keeps the valuable details hidden inside, allowing only the necessary bits to be accessed. It protects the inner workings, making the object a well-guarded secret.
  + is like a magic box. It wraps up the workings of an object, keeping the spells inside hidden. For example, a coffee machine encapsulates the process of brewing coffee—users just press a button, unaware of the intricate mechanisms inside.
  + is like a safe-deposit box, securing valuable information. It bundles data and methods together, protecting them from unauthorized access.
  + hides the internal details of a class, preserving the integrity of its implementation.

## Abstraction

* represents real-world objects in a simplified manner, focusing on essential features, like an **Interface**.
* A screenshot of a computer program

  Description automatically generatedis the art of simplification. It shows you only what you need to see and hides the complexities. It's like using a TV remote without knowing the intricate details of how it works, revealing only essential features —just press a button, and it does its job. It simplifies complexities, providing a user-friendly interface.
* is the art of the storybook. It tells the essential parts of a tale, omitting unnecessary details. Consider a car dashboard—it abstracts complex engine operations into simple indicators like speed and fuel level, allowing drivers to focus on critical information.

## Inheritance

* A computer error message

  Description automatically generatedA computer code with text

  Description automatically generatedis the family tree of classes, passing down characteristics. It lets a new object inherit traits from an existing one, creating a generational link, passing down characteristics. It's akin to a child inheriting certain features, properties and behaviors from their parents, creating a connection between the old and the new.
* Think of a smartphone inheriting features like a camera and touchscreen from its ancestor, the mobile phone.
* It fosters code reuse and establishes relationships.

## Polymorphism

* A screenshot of a computer code

  Description automatically generatedA screenshot of a computer program

  Description automatically generatedis an actor with many faces. It allows an entity to take on different forms or behaviors based on the situation. It's like a superhero adapting to various challenges—different tasks, same hero.
* A screenshot of a computer code

  Description automatically generatedis the chameleon of code. It adapts to different situations, taking on multiple forms. Imagine a shape-shifting robot—it can act as a car on roads, a boat in water, or a plane in the air, showcasing polymorphic behavior.

# OOP Concepts

## A screenshot of a computer code Description automatically generatedA computer code with text Description automatically generated with medium confidenceClass

* is the blueprint for objects, defining their structure and behavior.
* is like a blueprint for creating cars, it outlines the structure (properties) and behavior (methods) that each car of that type will have.
* It’s for creating intelligent, reusable entities in a programming world. It encapsulates data and behavior, serving as a template for creating objects with shared characteristics.

### A screenshot of a computer code Description automatically generatedBase Class:

* + like the wise elder in a family tree, providing shared traits, wisdom and behaviors that can be inherited by its specialized descendants. It serves as a foundation, offering a common structure **for derived classes to inherit and build upon.**
  + like the root guide in a forest, providing fundamental knowledge and a path for all creatures to follow. Each creature may have its traits, but they all share a connection to this common source.

A screenshot of a computer code

Description automatically generated

*In this analogy, the* ***Animal*** *class is the base class, offering common knowledge (Name) and a shared behavior (Eat method). Specialized creatures, like Cat or Dog, can inherit from this base class and express their unique characteristics.*

*Here, the* ***Cat*** *class inherits from the Animal base class, gaining common knowledge and shared behavior, while introducing its own distinct behavior (Meow method).*

*This creates a specific cat instance with both the common knowledge from the base class and the unique traits from the derived class -* ***Cat****.*

### Derived Class:

* is like a specialized room in the building, inheriting features from the base class and adding its unique characteristics.
* is like a unique manifestation in a series, taking inspiration from a common theme but expressing individuality and distinctive features.
* It extends and customizes the attributes and behaviors of a base class, adding unique features while leveraging the existing foundation.
* See Example above – **Cat class** being the derived class.

### A screenshot of a computer code Description automatically generatedAbstract Class:

* is like a blueprint with unfinished rooms, providing a template for derived classes to implement specific details.
* A black text on a white background

  Description automatically generatedA close-up of a text

  Description automatically generatedlike an unfinished script, providing a partial blueprint for classes to complete. It allows the definition of common structure and methods while leaving certain details to be implemented by derived classes. It may contain abstract methods and accessors.
* **An abstract class cannot be instantiated**.



* It is not possible to modify an abstract class with the sealed modifier because the two modifiers have opposite meanings.

A screenshot of a computer code

Description automatically generated

A black text on a white background

Description automatically generated

### A computer code with text Description automatically generatedSealed Class:

* is like a finalized contract, indicating **that a class cannot be extended or inherited**. It serves as a complete, standalone entity, preventing further modification or specialization.
* It's an intentional decision to conclude the development, ensuring that no further additions or modifications can be made.
* A close up of a clock

  Description automatically generatedA white background with black and red text

  Description automatically generatedA screenshot of a computer code

  Description automatically generatedA sealed class, in C#, is a class **that cannot be inherited** by any class but can be instantiated. The design intent of a sealed class is to indicate that the class is specialized and there is no need to extend it to provide any additional functionality through inheritance to override its behavior.

A black and white text

Description automatically generated

### A screenshot of a computer code Description automatically generatedStatic Class:

* + A black and white text

    Description automatically generatedA static class is like a communal meeting space, where members share common resources and services. It serves as a container for methods and properties that are universally accessible that cannot be instantiated or derived from. It serves as a central hub for functionalities that belong to the class itself rather than specific instances.
  + Static classes **cannot contain an instance constructor**. However, they **can contain a static constructor.** Non-static classes should also define a static constructor if the class contains static members that require non-trivial initialization.

A screenshot of a computer code

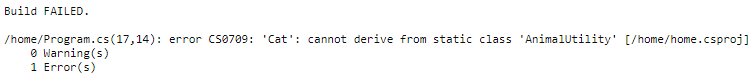
Description automatically generated

A black and white text

Description automatically generated

A black and white text

Description automatically generated



A screenshot of a computer code

Description automatically generated

### Partial Class:

* + is like dividing a large room into sections, allowing its implementation to be spread across multiple files.
  + like a collaborative effort, allowing a **class definition to be split across multiple files**. It enables developers to organize and extend a class's implementation in separate parts while presenting a unified view during compilation. It's a way to break down complexity, promoting teamwork in building a cohesive whole.

A computer screen shot of a person

Description automatically generated

A black and white text

Description automatically generated

### Generic Class:

* A generic class is like a versatile blueprint that **can work with different data types**.
* A black text on a white background

  Description automatically generatedA screen shot of a computer code

  Description automatically generatedA computer code with text

  Description automatically generatedIt can accommodate various data types without compromising design integrity. Allowing the creation of classes and methods with placeholders, fostering flexibility in implementation.

Here, the Box class is instantiated with both string and int types, showcasing the flexibility of a generic blueprint accommodating diverse data types.

In this example, the Box class is generic, denoted by the <T> syntax. It acts as a versatile container, allowing storage and retrieval of content of any type. This promotes adaptability without sacrificing the coherence of the class structure.

### Inner Class:

* An inner class is a class defined within another class, encapsulating functionality that is closely tied to the outer class. It helps in organizing and modularizing code.
* is like a hidden companion, residing within another class, providing a way to encapsulate functionality intimately tied to its containing class. It fosters a more organized and modular design by allowing the definition of supporting structures within the scope of a primary class.
* Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
* A private nested class can be used for a sub-object that is intended to be used only inside the parent. An example of this would be if a HashTable class contains a private Entry object to store data internally only.

A computer code with text

Description automatically generated



In this example, **CatHelper** is an inner class within the Animal class, encapsulating functionality related to making cat sounds. The inner class promotes a cleaner design by keeping the cat sound functionality closely associated with the Animal class.

### Singleton Class:

* A singleton class ensures only one instance is created and provides a global point of access. It's like a solitary guardian ensuring that only a single instance needs to coordinate actions across the application.
* The singleton pattern is one of the best-known patterns in software engineering. Essentially, a singleton is a class which only allows a single instance of itself to be created, and usually gives simple access to that instance. Most commonly, singletons don't allow any parameters to be specified when creating the instance - as otherwise a second request for an instance but with a different parameter could be problematic! (If the same instance should be accessed for all requests with the same parameter, the factory pattern is more appropriate.)
  + A single constructor, which is private and parameter-less. This prevents other classes from instantiating it (which would be a violation of the pattern). Note that it also prevents subclassing - if a singleton can be subclassed once, it can be subclassed twice, and if each of those subclasses can create an instance, the pattern is violated. The factory pattern can be used if you need a single instance of a base type, but the exact type isn't known until runtime.
  + The class is sealed. This is unnecessary, strictly speaking, due to the above point, but may help the JIT to optimize things more.
  + A static variable which holds a reference to the single created instance, if any.
  + A public static means of getting the reference to the single created instance, creating one if necessary.

A screenshot of a computer program

Description automatically generated

In this example, the ‘**Cat**’ class ensures the existence of a single instance. The private constructor and a method (GetInstance()) control access, guaranteeing that only one ‘**Cat’** roams the digital landscape.



### Immutable Class:

* An immutable class is like a sealed document, once created, its state cannot be changed. It ensures that instances remain constant, promoting predictable and thread-safe behavior, and eliminating the possibility of modification.
* To be immutable, all your properties and fields should be read-only. And the items in any list should themselves be immutable.
* **No Setters:** Remove public setters for properties, ensuring that the values of properties cannot be changed once the object is created.
* **Read-Only Properties:** Declare properties as read-only (get only) to prevent modification.
  + All properties should be getters only.
* **Initialization Through Constructor:** Initialize all properties in the constructor and make sure they are set only once during object creation.
  + All information injected into the class should be supplied to the constructor.
* **No Mutable References:** If the class contains reference types (objects), ensure that those objects are also immutable or are not externally modifiable.
  + If a collection (or Array) is passed into the constructor, it should be copied to keep the caller from modifying it later.
  + public readonly object[] MyObjects;=> is not immutable even if it's marked with **readonly** keyword. You can still change individual array references/values by index accessor.
* **Deep Copy or Defensive Copy:** If dealing with mutable objects within the class, consider creating deep or defensive copies to prevent external modifications.
  + If you're going to return your collection, either return a copy or a read-only version (for example, using [ArrayList.ReadOnly](http://msdn.microsoft.com/en-us/library/system.collections.arraylist.readonly.aspx) or similar - you can combine this with the previous point and store a read-only copy to be returned when callers access it), return an enumerator, or use some other method/property that allows read-only access into the collection
* **Seal the Class (Optional):** To prevent further derivation, you can use the sealed keyword in C#.

A screenshot of a computer program

Description automatically generated

In this example, the **Cat** class is designed as immutable with properties having private setters. Once a Cat object is created, its state (name, color, and age) remains fixed, promoting a stable and thread-safe design. The immutability ensures that instances of the class cannot be altered after creation, contributing to more predictable and functional code.

A close up of a number

Description automatically generated

## Object

* + An **object is an instance of a class.** It is an instance that encapsulates both state (data) and behavior, embodying the characteristics and actions defined by its class.
  + It represents a concept or entity in the given business domain the program resides in. The data it carries describes the characteristics or properties of said entity. The object’s behavior represents actions that can be operated on its data, and it’s expressed to external objects via the object’s methods.
  + like a dynamic entity brought to life from the blueprint of a class. It is a real-world representation within a program, capable of interacting with other objects and performing tasks.

A screenshot of a computer code

Description automatically generatedA screenshot of a computer code

Description automatically generated

In this example, **myCat** is an object instantiated from the Cat class, embodying the characteristics and behaviors defined in both Cat class and its base class Animal. Objects in OOP represent concrete instances that can perform actions and hold specific data.

Figure 1.1

## Method

* + A method is a **set of instructions bundled together within a class** to perform a specific task. It defines a specific behavior that an object can perform. It encapsulates executable code associated with an object, enabling it to carry out tasks and operations.
  + A method is like a behavioral script within a class, outlining specific actions an object can perform. It's a set of instructions guiding the object in expressing its unique behaviors.

In example above **Figure 1.1**, **Eat** is a method defined in the Animal class, representing a behavior associated with consuming food. Similarly, **Meow** is a method specific to the Cat class, representing the behavior of producing a sound. Methods provide a way for objects to perform actions and contribute to the overall functionality of a class.

## A computer code with text Description automatically generatedProperty

* + A **property is a characteristic of an object**, encapsulating its state.
  + like a dynamic attribute associated with an object, encapsulating data and often includes get and set methods, allowing manipulation and retrieval of the object's characteristics.



In this example, **Name is a property** defined in the Animal class. It encapsulates the private field \_name and provides controlled access through get and set methods. Properties are essential for managing the state of objects and ensuring encapsulation in object-oriented programming.

A close-up of a computer screen

Description automatically generated

# Different Design Patterns

## CQRS (Command Query Responsibility Segregation):

* CQRS is like having separate kitchens for cooking and cleaning dishes in a restaurant. It separates the command (write) operations from query (read) operations, optimizing the system for different tasks.
* It fosters a clearer, more tailored design by acknowledging that reading and writing data have distinct requirements. This means that functions for reading and writing data are not kept in the same interface or class. The main advantages of doing this include:
  + Different teams have the flexibility to handle these operations independently.
  + Each operation can be scaled based on its specific requirements. Write operations are typically utilized far less frequently than read operations.
  + Each operation can implement security measures tailored to its specific needs.
  + Read operations may adopt a distinct architecture to facilitate caching and conversions to data transformation objects, aligning with client requirements.
  + Write operations encompass tasks such as data validation and lookups.
* However, do keep in mind that this pattern is better suited to larger applications where the requirements and load levels between read and write operations are different. For a simple and small application, the normal CRUD pattern, often auto generated from ORM tools, is sufficient.

A screenshot of a computer program

Description automatically generated

In this example, the Animal class embodies the CQRS pattern. The **UpdateDetails method acts as a command**, changing the state of the object, while the **GetDetails method serves as a query**, retrieving the state. This separation of concerns enhances clarity and modularity in the design.

A black and white text

Description automatically generated with medium confidence

## Circuit Breaker

* A design pattern used in software development to improve the resilience and fault tolerance of applications that interact with external services or resources.
* A circuit breaker is like a safety switch for an application. It helps prevent cascading failures by temporarily blocking requests to a failing service, giving it time to recover.
* It monitors operations and, if a threshold of failures is reached, it suspends further execution. This helps prevent cascading failures and provides a way to handle faults gracefully.

A screenshot of a computer program

Description automatically generated

A black text on a white background

Description automatically generatedA computer screen shot of a code

Description automatically generated

## A screenshot of a computer program Description automatically generatedAggregator

* + An aggregator is like a chef preparing a diverse menu by combining ingredients from different sources. It consolidates data or results from multiple services to provide a unified response.
  + This pattern simplifies complexity and enhances flexibility by creating a single-entry point for diverse data.



## Decorator

* A decorator is like adding layers of toppings to a pizza without changing its base. It dynamically extends the behavior of an object without altering its structure.
* Decorator is a structural pattern that allows adding new behaviors to objects dynamically by placing them inside special wrapper objects, called decorators. Using decorators, you can wrap objects countless number of times since both target objects and decorators follow the same interface.
* It envelops entities with layers of functionality, allowing for fluid, runtime composition. It's a creative brushstroke that adorns the core-essence without altering its fundamental nature.

## Specification

* The Specification pattern is like a rulebook, encapsulating criteria for selecting or filtering objects. It provides a way to define and compose conditions, enabling the creation of expressive and reusable specifications to evaluate whether an object meets certain criteria.
* The Specification Pattern in .NET is akin to discerning criteria that encapsulate rules for object selection. It acts as a set of guidelines, enabling the creation of expressive conditions for querying and filtering entities. It's a language for articulating intricate conditions that transcend mere Boolean checks.

# Dependency Injection (DI) in .NET:

* Dependency Injection in .NET is like a chef delegating specific tasks to specialized sous-chefs. It's a technique where a class receives its dependencies rather than creating them, promoting modularity and making components more interchangeable. It's akin to assembling a dish with pre-prepared ingredients, allowing for flexibility and maintainability.
* It dynamically establishes connections within a software framework. It involves providing dependencies to objects externally, empowering them to seamlessly interact without the burden of creating or managing their dependencies.

A screenshot of a computer program

Description automatically generated

In this example, the **AnimalService** class demonstrates Dependency Injection by receiving an IAnimal dependency through its constructor. This promotes flexibility and maintainability, as the specific implementation of IAnimal (e.g., Cat) can be injected at runtime, allowing for easy testing and interchangeability of components.

A black text on a white background

Description automatically generated

## Types of Dependency Injection:

### A computer screen shot of text Description automatically generatedConstructor Injection:

* + Constructor injection is like handing a chef a set of essential tools when they start cooking. Dependencies are injected through the class's constructor, ensuring they are available when the object is created.
  + establishes a dynamic link between components at the time of creation. It involves passing dependencies through the constructor, ensuring that the required connections are secured right from the start.

### Property Injection:

* + A screenshot of a computer code

    Description automatically generatedProperty injection is akin to giving a chef an additional tool during the cooking process.
  + This is like adding optional decorations to a completed structure. Dependencies are injected through public properties of the class, allowing for flexibility in setting dependencies after the object is constructed.
  + Property Injection is akin to dynamically linking components through designated properties. Dependencies are injected post-construction, allowing for flexibility in establishing connection.

### A computer screen shot of a program code Description automatically generatedMethod Injection:

* + Method injection is like a chef asking for specific tools when needed.
  + Dependencies are injected through method parameters, enabling specific operations to receive the required dependencies. It allows for on-the-fly connection establishment during method invocation.

### When to Use Dependency Injection:

* + Use DI when you want to promote modularity and testability in your application.
  + Use DI when you want to decouple components, making it easier to replace or extend functionality.
  + Use DI when you want to manage the lifecycle of dependencies and handle their creation externally.

# Reserved Words / References

## Delegates

* Delegates in .NET are like event organizers. They define a type that represents references to methods, enabling a flexible way to pass and invoke functions.
* A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime.
* Delegates are used to define callback methods and implement event handling,
* These are used to represent or refer to one or more functions.
* These can only be used to define call-back methods.
* If we want to chain together multiple methods or sequential processing and call via single event, then delegates are good approach.
* Delegates provide abstraction and encapsulation over methods.
* To create publisher and subscriber model then delegates are good approach.
* This eliminates the need to unnecessarily use multiple inheritance via interfaces.
* If you are building a library that can be used by any developers, then if you code against delegates your code will be loosely coupled with the user's code and every other developer can use that method with their own function definition. In Short Open Closed Principle with Delegates.
* How to declare or Syntax of a delegate?
  + - Delegate is declared via 4 steps:  1) Declare 2) Create 3) Point 4) Invoke

A screen shot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

## Anonymous Functions

* A close up of text

  Description automatically generatedAnonymous functions are like pop-up performances.
* They allow you to create functions without explicitly naming them, enhancing conciseness and readability.
* Anonymous functions in .NET are like stealthy performers, executing tasks without a formal declaration.
* They are concise, ephemeral functions used on-the-fly

for tasks that don't warrant a full method.

## A screen shot of a computer program Description automatically generatedEvent

* In .NET, an event is a mechanism for communication between objects.
* It represents an occurrence or a notification that an object can subscribe to and respond to.
* Events are based on delegates but provide a higher level of encapsulation.
* Unlike functions or methods, events allow multiple subscribers (listeners) to react to an occurrence.

In this example, the Publisher class declares an event named MyEvent, and the Subscriber class has a method named HandleEvent that serves as an event handler. The Main method demonstrates how to subscribe the handler method to the event and trigger the event, resulting in the handler method being invoked. Unlike traditional method calls, events provide a decoupled and extensible way to handle notifications in the application.

## LINQ (Language Integrated Query)

* LINQ (Language Integrated Query) is like a versatile query language within C#.
* like a query alchemist, transforming data structures with a language-integrated approach. It provides a powerful syntax for querying and manipulating collections, bringing a seamless querying experience to C#.
* A computer code with text

  Description automatically generated with medium confidenceit is a uniform query syntax in C# and VB.NET to retrieve data from different sources and formats.
* LINQ queries return results as objects. It enables the use of object-oriented approach on the result set and not to worry about transforming different formats of results into objects.

## Lambda Expressions

* A close up of text

  Description automatically generatedLambda expressions are like shorthand notes for functions. They provide a concise syntax for defining anonymous methods, often used in LINQ and functional programming.
* Use a *lambda expression* to create an anonymous function.
* In lambda expressions, the lambda operator => separates the input parameters on the left side from the lambda body on the right side.
* Any lambda expression can be converted to a delegate type. The delegate type to which a lambda expression can be converted is defined by the types of its parameters and return value. If a lambda expression doesn't return a value, it can be converted to one of the Action delegate types; otherwise, it can be converted to one of the Func delegate types.

## A computer code with black text Description automatically generatedGenerics

* Generics are like customizable containers. The class, structure, or interface declaration functions as a template, with placeholders for the types that it can contain or use.
* It accommodates various data types without sacrificing type safety.

## Constructor & Deconstructor

### constructor

* + A constructor is a method used to initialize the state of an object, and it gets invoked at the time of object creation. **Rules** for constructor are:
    - Constructor Name should be the same as a class name.
    - A constructor must have no return type.

### destructor?

* + A destructor is a method which is automatically called when the object is made of scope or destroyed.
  + Destructor name is also same as class name but with the tilde (~) symbol before the name.

## CONST VS READONLY

### const

* + A white background with black and pink text

    Description automatically generatedconst is like engraving a command in stone; it creates a compile-time constant whose value is known at compile time and cannot be changed. It's suitable for values that are known and won't change throughout the program's execution.

### readonly

* + A computer code with black text

    Description automatically generatedreadonly is a runtime constant whose value can be set at runtime and remains constant throughout the object's lifetime. It's suitable for values that may be determined at runtime but won't change afterward.
  + A field whose value can only be assigned at the time of declaration or within the constructor. It creates a steadfast marker, ensuring that the value remains constant after initialization.

## VAR VS DYNAMIC

### var

* A **Type Inference**. It allows the compiler to infer the type of variable based on the assigned value during compile-time. It is not a dynamic type; rather, it is statically typed at compile-time.

### dynamic

* A dynamic type. Variables declared with dynamic are resolved at runtime, allowing for late binding and dynamic behavior.

## Heap, Stack, and Queue

* In summary, the **heap** is for dynamic memory allocation, the **stack** is for method call information and local variables, and a **queue** is a data structure for handling items in a First In, First Out order.

### Heap:

* Meaning: The heap is a region of a computer's memory used for dynamic memory allocation. In .NET, it is where objects are allocated and managed by the garbage collector.
* Example: When you create an object using new in C#, it is allocated on the heap.
* For example: var myObject = new MyClass();

### Stack:

* Meaning: The stack is a region of memory that stores method call information, local variables, and other function-related details. It follows a Last In, First Out (LIFO) structure.
* Example: Local variables and method call information are stored on the stack. For example:
* void MyMethod() { int localVar = 10; // localVar is stored on the stack }

### Queue:

* Meaning: A queue is a data structure that follows a First In, First Out (FIFO) order. In .NET, it can be implemented using the Queue<T> class in the System.Collections.Generic namespace.
* Example: Enqueueing and dequeuing items from a queue:
* Queue<int> myQueue = new Queue<int>();
* myQueue.Enqueue(1); // Enqueue
* int dequeuedItem = myQueue.Dequeue(); // Dequeue

## Static

* In .NET, the static keyword is used in various contexts—within a class, a method, or a property—and its meaning differs based on where it's applied.
* It cannot be used with indexers of finalizers.
* In summary, the static keyword denotes different behaviors based on its context:

### Static in a class:

* A static class is like a toolbox that doesn't need instantiation. It cannot be instantiated or have instance members, and its members must be static. It's commonly used for utility classes that provide functionality without the need for an instance.
* In a class, it designates a class as static, preventing instantiation.

### Static in a Method:

* A static method is like a stand-alone performer; it belongs to the class rather than an instance. It can be called without creating an instance of the class, and it cannot access non-static (instance) members directly.
* In a method, it indicates that the method is associated with the class rather than an instance.

### Static in a Property:

* A static property is like a shared piece of information across all instances of a class. It's associated with the class itself rather than with individual instances, and it can be accessed without creating an instance of the class.
* In a property, it signifies that the property is shared among all instances of the class and can be accessed without creating an instance.

## RESTful Api and principles

* RESTful API (Representational State Transfer) is an architectural style for designing networked applications, particularly web services. It follows a set of principles that promote simplicity, scalability, and statelessness. In .NET, you can implement RESTful APIs using ASP.NET Web API or ASP.NET Core.

### RESTful Principles:

#### Statelessness:

* Each request from a client to a server must contain all the information needed to understand and fulfill the request. The server should not store any information about the client's state between requests.

#### Client-Server Architecture:

* The client and server are separate entities that communicate over a stateless protocol. The client is responsible for the user interface, while the server is responsible for processing requests and managing resources.

#### Uniform Interface:

* A uniform and consistent set of interactions should be defined to simplify the architecture and improve scalability. Key principles include resource identification, resource manipulation through representations, and self-descriptive messages.

#### Resource-Based:

* Resources are identified by URIs (Uniform Resource Identifiers) and manipulated using standard HTTP methods (GET, POST, PUT, DELETE). Resources can be entities like users, products, or any other meaningful concept.

#### Representation:

* Resources are represented in a format (e.g., JSON, XML) that can be easily understood by clients. The client can modify the representation and send it back to the server for updates.
* Implementation in ASP.NET Core:

### Define a Controller:

* Create a controller that inherits from ControllerBase. Each action in the controller represents a different RESTful endpoint.

### Define Actions:

* Use attributes like [HttpGet], [HttpPost], [HttpPut], and [HttpDelete] to specify the HTTP methods for each action.

### Routing:

* Use routing attributes to define the URI templates for each action.

### Return Types:

* Actions should return appropriate HTTP status codes and data representations (JSON, XML) for success or failure.

### Content Negotiation:

* ASP.NET Core supports content negotiation, allowing clients to specify the desired content type in the request headers.
* Common return types can be used to respond to client requests in a RESTful manner.

### IActionResult:

* Represents the most generic result type. Allows you to return different HTTP status codes and content types.

### ActionResult<T>:

* Introduced in .NET Core 2.1, allows you to specify a return type. Provides a cleaner syntax for returning specific HTTP status codes and data.

### ObjectResult:

* Allows you to return a custom object along with an HTTP status code.

### NotFoundResult:

* Returns a 404 Not Found status.

### BadRequestResult:

* Returns a 400 Bad Request status.

### NoContentResult:

* Returns a 204 No Content status.

### CreatedAtActionResult:

* Returns a 201 Created status with a location header.

### FileResult:

* Returns a file to the client.

## OPENAPI Swagger configuration

* OpenAPI, formerly known as Swagger, is a specification for documenting and visualizing RESTful APIs. In .NET, Swagger can be integrated into your ASP.NET Web API or ASP.NET Core application to generate interactive API documentation.

### Implementation in ASP.NET Core:

* Install Swashbuckle NuGet Package:
* dotnet add package Swashbuckle.AspNetCore

### A screenshot of a computer program Description automatically generatedConfigure Swagger in Startup.cs:

* In the Startup.cs file, add the Swagger middleware and configure it.

### Documenting API Controllers:

* Add XML comments to your API controllers for additional documentation.

### Viewing Swagger UI:

* Run your application and navigate to /swagger or /swagger/index.html to access the Swagger UI. This interactive documentation allows users to explore and test your API.

## Garbage Collection

* Garbage Collection (GC) in .NET is a memory management process that automatically identifies and frees up memory occupied by objects that are no longer in use. The GC helps prevent memory leaks and enhances the overall stability and performance of .NET applications.

### Mark and Sweep Algorithm:

* During the marking phase, the garbage collector traverses the object graph, starting from known root objects (e.g., global variables, stack, and registers), marking all reachable objects.
* In the sweeping phase, the collector identifies and releases memory occupied by objects that were not marked, as they are considered unreachable and can be safely collected.

### Generational Collection:

* Objects are categorized into three generations (young, middle-aged, and old) based on their lifetime. Most objects die quickly, so the younger generation is collected more frequently, while older generations are collected less often.
* The generational approach improves efficiency by focusing on short-lived objects in the younger generation.
* In .NET, you don't explicitly manage memory deallocation; the GC handles it automatically. However, you can influence the GC process using the System.GC class.
* // Explicitly trigger garbage collection (for demonstration purposes)
* System.GC.Collect();
* After performing some work, System.GC.Collect() is called to explicitly trigger garbage collection. Note that in a real-world scenario, you generally don't need to manually invoke garbage collection.

## Abstract

* Abstract is like a blueprint, denoting a class or member without implementation details.
* It serves as a foundation for concrete implementations.

### Abstract Keyword in a Class:

* the abstract keyword is used to define an abstract class. An abstract class cannot be instantiated on its own and typically serves as a base class for other classes. It can contain abstract and non-abstract (concrete) methods.

### Abstract Keyword in a Method:

* In the context of a method, the abstract keyword is used to define an abstract method within an abstract class. Abstract methods have no implementation in the base class and must be implemented by any concrete (non-abstract) derived classes.

## Access Modifiers

* Access modifiers in C# determine the visibility and accessibility of classes, members, and types within the program. It allows developers to design encapsulated and modular code.
* An *access modifier* defines *who* can access the method or class, and *when* (i.e.: private: only class members, public: everyone else etc). Marking a method or a class as sealed means that it cannot be inherited. It says nothing about *access* per se.

### public

* + The public access modifier makes a class, method, or member accessible from any other class or assembly.
  + Public is like a code amphitheater, allowing unrestricted access to a type or member. It makes the type or member accessible from any other code.

### private

* + The private access modifier limits the visibility of a class, method, or member to only within its containing type (class or struct).
  + Like a code fortress, restricting access to a type or member within the same class. It enforces encapsulation and data hiding.

### protected

* + The protected access modifier allows access within the same class or derived classes.
  + Like a code citadel with limited access, allowing access limited to the same class or derived classes. It supports controlled sharing among related types.

### internal

* + The internal allows access within the same assembly but restricts access from external assemblies.
  + Like a code privacy shield, specifying that a type or member is accessible within the current assembly but not outside. It restricts visibility to a defined scope.

### protected internal

* + The protected internal access modifier allows access within the same assembly or derived classes, regardless of assembly boundaries.

### private protected

* + The private protected access modifier allows access within the same assembly and only in derived classes within the same assembly.

## Data Types

### Value types

#### Integral Types

##### byte

* + - byte is like a tiny data morsel, representing an 8-bit unsigned integer. used to declare variables that store integer values between 0 and 255. It is an unsigned integer type that takes up 1 byte of memory.

##### int

* + - int is like a numeric workhorse, representing a 32-bit signed integer. It's a fundamental data type for storing whole numbers.

##### long

* + - long is like a numeric stalwart, representing a 64-bit signed integer. It accommodates a wide range of whole number values.

##### short

* + - short is like a concise numeric envoy, representing a 16-bit signed integer. It accommodates a range of small to moderate integer values.

##### sbyte

* + - sbyte is like a compact numeric courier, representing an 8-bit signed integer. It's suitable for small integer values.

##### ushort

* + - ushort is like a concise numeric herald, representing a 16-bit unsigned integer. It's suitable for small positive integer values.

##### uint

* + - uint is like an unsigned numeric envoy, representing a 32-bit unsigned integer. It accommodates a range of positive integer values.

##### ulong

* + - ulong is like a vast numeric ambassador, representing a 64-bit unsigned integer. It accommodates a wide range of large positive integer values.

#### Floating-Point Types

##### float

* + - float is like a nimble numeric companion, representing a single-precision floating-point type. It's suitable for a broad range of numeric values.

##### double

* + - double is like a versatile numeric palette, representing a double-precision floating-point type. It's suitable for a wide range of numerical values.

##### decimal

* + - decimal is like a precise numeric artisan, representing a floating-point type with extended precision. It's suitable for financial calculations requiring accuracy.

#### Character Type

##### char

* + - char is like a single character ambassador, representing a Unicode character in .NET. It's essential for working with individual characters in strings.

#### Boolean Type

##### bool

* + - bool is like a binary sentinel, representing a Boolean type with values of true or false. It's fundamental for logical conditions and decision-making.

#### Enumeration Type

##### enum

* + - Enum is like a symbolic labeler, defining a set of named integral constants. It enhances code readability by providing meaningful names to numeric values.

### Reference Types

#### Object Type

##### object

* + - The base type for all .NET types. It can hold any value.

#### String Type

##### string

* + - string is like a textual maestro, representing a sequence of characters. It's a fundamental data type for working with text in .NET. It represents a sequence of characters.

#### Array Type

##### array

* + - Represents a collection of elements of the same type.

#### Class Types

##### class

* + - User-defined reference type.
    - Class is like a code container, encapsulating data and behavior. It serves as a blueprint for creating objects, facilitating code organization and reuse.

#### Interface Type

##### interface

* + - Defines a contract for implementing classes.
    - Interface is like a contract enforcer, defining a set of method signatures that implementing classes must adhere to. It facilitates polymorphism and code organization.

#### Delegate Types:

##### delegate

* + - Represents a method signature.
    - Like a dynamic messenger, representing a reference to a method. It allows for flexible method invocation and event handling.

## Conditional Statements

* it allows developers to write flexible and expressive code that responds to various conditions during program execution.
* The choice of which to use depends on the specific requirements of the application logic.

### **if** statement:

* + The if statement allows you to execute a block of code if a specified condition evaluates to true.
  + It’s like a gatekeeper, determining whether a certain block of code should be executed based on a specified condition. It's fundamental for decision-making.

### **else** statement:

* + The else statement is used in conjunction with an if statement to specify a block of code that should be executed if the condition in the if statement is false.
  + is like an alternative path, specifying the code to be executed when the condition in an if statement is false. It provides an alternative route in decision-making.

### **else if** statement:

* + The else if statement is used to test multiple conditions sequentially. It follows an if statement and precedes an optional else statement.

### **switch** statement:

* + The switch statement provides a way to handle multiple conditions based on the value of an expression. It's a more concise way to write multiple if and else if statements. It simplifies complex branching logic.

## Comparison Mechanisms

* In C#, there are various mechanisms for comparison, each serving different purposes and contexts.
* These comparison mechanisms cater to different scenarios and types of objects, providing flexibility and customization in handling equality and ordering. The choice of mechanism depends on the specific requirements of the comparison operation.

### Equality Operator (**==**):

* + Usage: Used for comparing values for equality.
  + Behavior: Compares values for equality and returns a Boolean result.

### Inequality Operator (**!=**):

* + Usage: Used for comparing values for inequality.
  + Behavior: Compares values for inequality and returns a Boolean result.

### Equals Method:

* + Usage: Used to compare objects for equality.
  + Behavior: The Equals method is a virtual method defined in the Object class, and it's often overridden in custom classes to provide custom equality semantics.

### Reference Equality (ReferenceEquals):

* + Usage: Used to check if two references point to the same object.
  + Behavior: Compares references for identity, not the content of objects.

### String Comparison (String.Equals with StringComparison):

* + Usage: Used to compare strings with different comparison options.
  + Behavior: Allows for case-sensitive or case-insensitive string comparisons.

### CompareTo Method (for Comparable Types):

* + Usage: Used by types implementing IComparable for natural ordering.
  + Behavior: Compares the current instance to another instance and returns an integer indicating the relative order.

### Comparer Default (Comparer.Default):

* + Usage: Used for comparing objects using their default comparer.
  + Behavior: Allows comparing objects without specifying a specific comparer.

## Conditional Operators

* In .NET, conditional operators are used to perform conditional evaluations and expressions. Here are some of the commonly used conditional operators:
* These operators play a crucial role in writing expressive and concise conditional logic in C# and .NET applications. The choice of operator depends on the specific requirements of the condition being evaluated.

### Conditional Operator (? : - Ternary Operator):

* + Usage: Creates a concise conditional expression.
  + Syntax: condition ? expressionIfTrue : expressionIfFalse;

### Null Coalescing Operator (??):

* + Usage: Provides a default value when encountering a null reference.
  + Syntax: nullableValue ?? defaultValue;

### Null Conditional Operator (?.):

* + Usage: Safely accesses members of an object that may be null.
  + Syntax: object?.Member;

### Coalesce Assignment Operator (??=):

* + Usage: Assigns a value only if the variable is null.
  + Syntax: variable ??= value;

### Logical AND (&&):

* + Usage: Returns true if both operands are true.
  + Syntax: condition1 && condition2;

### Logical OR (||):

* + Usage: Returns true if at least one of the operands is true.
  + Syntax: condition1 || condition2;

### Logical NOT (!):

* + Usage: Returns the opposite of the operand's value.
  + Syntax: !condition;

## Looping or Iteration

* In .NET, there are several ways to perform looping or iteration to repeat a set of statements. Some common looping constructs in C#:

### **‘for’** loop:

* is like a controlled countdown, executing a block of code a specified number of times.
* It's a fundamental construct for repetitive tasks.

### **‘while’** loop:

* is like a continuous check, repeatedly executing a block of code as long as a specified condition is true.
* It's fundamental for iteration.

### **do-while** loop:

* Like while loop, but the block of code is executed at least once before checking the condition.

### **foreach** loop:

* foreach is like a collection navigator, iterating over elements in a collection. It simplifies the process of traversing arrays or collections.

### **break** statement**:**

* break is like an exit command, breaking out of loops or switch statements. It disrupts the normal flow of execution and moves to the next statement.

### **continue** statement:

* Continue is like a skip command, advancing to the next iteration in a loop. It allows for the exclusion of specific statements within the loop.

### **goto** statement:

* Goto is like a code teleporter, allowing a direct jump to a labeled statement within a program. It's a rarely used construct, often discouraged for maintaining code clarity.

### **endLoop**:

* These looping constructs provide flexibility for different scenarios, and their usage depends on the specific requirements of your code.

## Asynchronous Programming

* Asynchronous programming in .NET allows executing non-blocking operations, improving application responsiveness.
* Async programming in .NET is like having a multitasking assistant. It allows your program to efficiently handle tasks that might take time, ensuring it doesn't get stuck waiting. It's akin to delegating chores to someone while you continue with other work.
* A screen shot of a computer program

  Description automatically generatedIt's achieved using the **async** and **await** keywords.

**async** keyword:

* Applied to a method, indicating that the method contains asynchronous operations.

**await** keyword:

* Used within an async method to asynchronously wait for the completion of a task or operation.

In this example, DownloadContentAsync is an asynchronous method that uses await to asynchronously download content from a URL. The Main method calls this asynchronous method, allowing the program to perform other tasks while waiting for the download to complete.

## Multi-threading

* A screenshot of a computer program

  Description automatically generatedMulti-threading in .NET is like orchestrating a team of synchronized dancers. It allows your program to seamlessly execute multiple tasks concurrently, akin to each dancer performing distinct movements in harmony, enhancing overall performance and responsiveness.
* Multithreading in .NET involves executing multiple threads concurrently to improve application performance and responsiveness. It allows parallel execution of tasks. Threading is achieved using the Thread class or higher-level constructs in the System.Threading namespace. Here's a concise example using the Thread class:

In this example, two threads (thread1 and thread2) are created and started to execute the DoWork method concurrently with the main thread. The output demonstrates interleaved execution between the main thread and the worker threads. The Join method ensures that the main thread waits for both worker threads to finish before proceeding.

## ASP.NET Core MVC Filte Execution sequenceAction Filters

* Action filters are attributes that can be applied to controllers or controller methods to modify the behavior of the action.
* Are like vigilant security guards at the entrance of a building. They inspect and influence incoming requests or outgoing responses to ensure specific conditions are met.
* These filters enable you to add custom behavior before or after an action method is executed, enhancing the overall control and security of the application.
* You can override the methods in your controller class.

| Filter type | Implemented by |
| --- | --- |
| Authentication filter | IAuthenticationFilter |
| Authorization filter | IAuthorizationFilter |
| Action filter | IActionFilter |
| Result filter | IResultFilter |
| Exception filter | IExceptionFilter |

### Types of Action Filters:

#### Authorization Filters:

* + Authorization filters act as access control personnel, determining whether a user has the right permissions to access a specific action method.
  + Authorization filters, if any, run first. This includes filters like [Authorize].
  + A screenshot of a computer error

    Description automatically generatedRestricts access to the action to authenticated users.
  + Authentication in ASP.NET Core is often configured in the Startup.cs file using authentication middleware. You can specify authentication schemes, configure external authentication providers, and set up various authentication options in the ConfigureServices and Configure methods of the Startup class
  + While there isn't a dedicated "Authentication" attribute, the [Authorize] attribute, in combination with authentication middleware, provides robust authentication capabilities in ASP.NET Core.

#### Resource Filters:

* + Resource Filters are mostly used for Logging, Caching, Throttling, Modifying model binding, and run next.
  + We have 2 types of Resource filter one is Sync and another is Async, for implementing Sync type we need to inherit IResourceFilter and for implementing Async type we need to inherit IAsyncResourceFilter Filter.
  + IResourceFilter
    - OnResourceExecuting - This method is executed before the associated action method is invoked.
    - OnResourceExecuted - This method is executed after the associated action method has been invoked.
  + IAsyncResourceFilter
    - OnResourceExecutionAsync - allows you to execute asynchronous code before the action method and to await the completion of subsequent filters or the action method.

#### Action Result Filters:

* + Action result filters are like post-event coordinators. They inspect and modify the result of an action method before it's sent to the client.
  + Action Filters executes just before and after execution of Action Method.
  + The most common use of action filters is to validate ASP.NET Core Model using ValidationAttribute or tracking user activity. Or the DataAnnotations.
  + Action filters, implementing IActionFilter or IAsyncActionFilter, are executed before and after the action method.
  + Some of the attributes are (**ValidateAntiForgeryToken, OutputCache**

#### Exception filters

* + act as emergency responders. They catch and handle exceptions that occur during the execution of an action method.
  + The attribute HandleAttribute is one of those; it Handles errors by redirecting to a specified error view.

#### Result Filter

* + Result filters are commonly used for tasks such as logging, modifying the result, or handling exceptions that occur during the execution of the action result.
  + Runs only after the controller action method executed successfully.
  + IResultFilter: This interface provides synchronous methods for result filter execution.
  + OnResultExecuting: Executed before the action result is executed.
  + OnResultExecuted: Executed after the action result is executed.
  + IAsyncResultFilter: This interface provides asynchronous methods for result filter execution.
    - OnResultExecutionAsync: Executed asynchronously before and after the action result is executed.

## Middleware in C# 6.0

* Middleware in C# 6.0 is like a skilled translator at a global conference. It sits between different components, facilitating communication and performing tasks such as translating messages or enriching content.
* It refers to components in the ASP.NET Core pipeline that handle HTTP requests and responses. Middleware components are arranged in a sequence to process requests and responses as they flow through the application.
* are responsible for tasks such as request processing, logging, authentication, and more. They form a pipeline where each component in the pipeline can inspect, modify, or pass along the request and response.
* It is added to the pipeline in the Configure method of Startup.cs. Middleware components can perform operations before and after the next component in the pipeline, allowing for flexible request and response processing.

## .NET 6.0 Out-of-Box Features

* These features enhance the robustness, maintainability, and security of your .NET 6.0 applications by providing convenient, built-in solutions for common challenges.

### Logging:

* + Logging in .NET 6.0 is like having a detailed diary for your application's journey. It captures important events and messages during runtime, helping developers understand what's happening under the hood.
  + Example Code:

### Error/Exception Handling:

* + Exception handling in .NET 6.0 is like a safety net for your code. It allows you to gracefully catch and manage errors, preventing your application from crashing unexpectedly.

### Caching:

* + Caching in .NET 6.0 is like having a shortcut to fetch frequently used information. It stores data temporarily, reducing the need to recalculate or retrieve it from the original source.
  + Feature management in .NET 6.0 is like having a switchboard for your application's functionalities. It allows you to toggle features on or off dynamically, providing flexibility in controlling what your application can do.

### Secret Management:

* + Secret management in .NET 6.0 is like storing sensitive information in a secure vault. It provides a way to securely store and access confidential data such as API keys, connection strings, or passwords.

## Feature-Based Development in .NET:

* Feature-based development in .NET is like constructing a modular building with individual wings.
* refers to an approach where the development and organization of code are structured around specific features or functionalities of an application. Instead of organizing code strictly by layers (such as models, controllers, and views in MVC), feature-based development groups related code and components together based on the features they implement.
* Like having separate rooms for different purposes within a house, allowing for independent development, testing, and deployment of specific features.
* Here are key aspects of Feature-Based Development:

### Organizing Code by Features:

* + Code related to a particular feature is grouped together. This includes controllers, views, models, services, and any other components associated with that feature.

### Encapsulation of Features:

* + Features are encapsulated and isolated from each other. This encapsulation makes it easier to understand, maintain, and extend specific functionalities without affecting the entire application.

### Clearer Code Structure:

* + The code structure reflects the organization of features in the application. This can result in a clearer and more modular codebase.

### Reusability:

* + Components related to a feature can be potentially reused in other parts of the application or even in different projects.

### Independent Deployment:

* + Features can be developed and deployed independently. This modularity facilitates continuous integration and continuous delivery (CI/CD) practices.

### Maintainability:

* + Feature-based development can enhance maintainability, as developers can focus on specific features without having to navigate through unrelated code.

## ORM, ADO.NET and the Entity Framework

### ORM (Object-Relational Mapping):

* + Meaning: ORM is a programming technique that allows data to be manipulated and queried using an object-oriented paradigm while being stored in a relational database.
  + Example: In an ORM, a class in an object-oriented language represents a table in a relational database. For instance, Entity Framework (EF) is an ORM framework in .NET.

### ADO.NET (ActiveX Data Objects for .NET):

* + Meaning: ADO.NET is a set of libraries and technologies in .NET for data access. It provides a set of classes for connecting to databases, executing queries, and managing results.
  + Example: ADO.NET includes classes like SqlConnection for database connections, SqlCommand for executing SQL queries, and SqlDataReader for reading query results.

### Entity Framework (EF):

* + Meaning: EF is an ORM framework in .NET that simplifies data access by enabling developers to work with databases using .NET objects. It provides a high-level abstraction over the database.
  + Example: With EF, you can define a C# class (an entity) that corresponds to a database table. For example:

## GitHub Keywords, Methods, and Actions:

### Repository:

* + A repository is like a shared workspace where collaborators gather to contribute and manage project files. It's akin to a virtual office for your codebase.
  + Example Code (Creation): *git init*

### Clone:

* + Cloning is like setting up a satellite office. It copies a repository to your local machine, allowing you to work on the project without disrupting the main office.
  + Example Code: *git clone https://github.com/username/repo.git*

### Branch:

* + A branch is like creating a parallel universe within your project. It enables you to work on new features or fixes without affecting the main timeline.
  + Example Code (Creation and Switching): *git branch new-feature git checkout new-feature*

### Commit:

* + Committing is like saving your work in a project journal. It captures a snapshot of changes, providing a historical record of your contributions.
  + Example Code:  *git add . git commit -m "Add new feature"*

### Pull Request:

* + A pull request is like submitting a proposal to the main office. It suggests changes made in your branch, requesting the project manager to review and merge.
  + Example Code (GitHub Interface*): # Open a pull request after pushing your branch*

### Merge:

* + Merging is like consolidating updates from multiple satellite offices. It combines changes from one branch into another, creating a unified codebase.
  + Example Code (GitHub Interface or Command Line): *git merge feature-branch*

### Fork:

* + Forking is like creating your own branch office from an existing project. It duplicates a repository, allowing you to experiment and make changes independently.
  + Example Code (GitHub Interface): *# Fork a repository from the GitHub interface*

### Star:

* + Starring is like giving a project a gold star. It shows appreciation or bookmarking a repository for future reference.
  + Example Code (GitHub Interface): *# Click the "Star" button on a repository*

## NuGet Package Creation and Version Maintenance:

### NuGet Package Creation in .NET:

#### Create a Class Library:

* + Start by creating a class library project in Visual Studio. This will be the project you want to package.

#### Configure Project Properties:

* + Set the assembly information and version in the project properties.

#### Build the Project:

* + Build the class library project to generate the necessary output.

#### A computer code with text Description automatically generatedCreate a NuGet Specification (.nupkg) File:

* + Create a .nupkg file that defines the metadata for your package. You can use the NuGet Command Line (nuget.exe) or create a .nuspec file.
  + Example .nuspec file:

#### Package the Project:

* + Use the NuGet CLI or Visual Studio to package your project. For example, using the command line:
  + nuget pack MyPackage.nuspec

#### Publish the Package (Optional):

* + If you want to share your package, you can publish it to a NuGet repository. Use the nuget push command.
  + nuget push MyPackage.1.0.0.nupkg -Source https://www.nuget.org/api/v2/package

## Azure DevOps Platform:

Pipeline:

* + Meaning: A pipeline defines the process for continuous integration and delivery (CI/CD) in Azure DevOps. It consists of stages, jobs, and tasks.

### YAML:

* + Meaning: YAML (Yet Another Markup Language) is a human-readable data serialization format. In Azure DevOps, it is commonly used to define build and release pipelines.

### Artifact:

* + Meaning: An artifact is a deployable component of your application. It could be a build output, package, or any other files needed for deployment.
  + Example: An artifact published from a build pipeline can be a ZIP file containing compiled binaries.

### Agent:

* + Meaning: An agent is a piece of software that runs tasks in a build or release pipeline. It can be a Microsoft-hosted agent or a self-hosted agent.
  + Example: A self-hosted agent installed on a specific machine to perform builds and releases.

### Release:

* + Meaning: A release in Azure DevOps represents the deployment of a specific version of your application to a target environment.
  + Example: Deploying version 1.0.0 of an application to the production environment.

### Variable:

* + Meaning: Variables in Azure DevOps are used to store and refer to values that can be used across tasks in a pipeline.

### Service Connection:

* + Meaning: A service connection is a link between Azure DevOps and an external service, like Azure, GitHub, or Docker, enabling secure communication.
  + Example: Configuring a service connection to access Azure resources in a pipeline.

### Task:

* + Meaning: A task is a single unit of work in a pipeline. It represents an action or a set of actions to be performed.
  + These keywords are fundamental in defining and configuring CI/CD pipelines in Azure DevOps. They play a crucial role in automating the build, test, and deployment processes.

## Splunk

* A screenshot of a computer program

  Description automatically generatedIs for log analysis, monitoring, and gaining insights from machine data. Understanding them is crucial for effective data querying and visualization in Splunk.

### Index:

* + Meaning: An index in Splunk is a repository for storing and retrieving events. It is the main storage mechanism for the data that Splunk processes.
  + Example: index=web\_logs

### Search Query:

* + Meaning: A search query in Splunk is used to retrieve and analyze data from indexed events.
  + Example: sourcetype=apache\_access status=200

### Event:

* + Meaning: An event in Splunk is a single log entry or data point that is stored in an index.
  + Example: timestamp=2022-01-01T12:00:00 source="/var/log/app.log" message="Error: Server unavailable"

### Field:

* + Meaning: Fields in Splunk represent key-value pairs extracted from the raw data. Fields are used for searching and analyzing data.
  + Example: status=404 method=GET

### Splunk Query Language (SPL):

* + Meaning: SPL is the query language used in Splunk for searching, analyzing, and visualizing data.
  + Example: index=web\_logs status=200 | stats count by host

### Dashboard:

* + Meaning: A dashboard in Splunk is a visual representation of data. It can include charts, tables, and other visualizations.
  + Example: A dashboard displaying server performance metrics.

### Alert:

* + Meaning: An alert in Splunk is a condition that, when met, triggers a notification. It helps in monitoring and responding to specific events.
  + Example: An alert configured to notify when the error rate exceeds a threshold.

### Source Type:

* + Meaning: Source type in Splunk identifies the format of the incoming data. It helps in parsing and extracting fields.
  + Example: sourcetype=apache\_access

### Indexer:

* + Meaning: An indexer in Splunk is responsible for storing and managing indexed data. It processes and makes the data available for searching.
  + Example: Configuring multiple indexers for distributed data storage.

### Forwarder:

* + Meaning: A forwarder in Splunk is a component that collects and sends data from sources to the Splunk indexers.
  + Example: Installing a Universal Forwarder on a server to forward logs to Splunk.

## Serilog

* Serilog is a .NET logging library that provides a flexible and structured approach to logging. It allows developers to log events with rich structured data and supports various sinks for log output.

## LaunchDarkly

* A computer screen shot of a computer code

  Description automatically generatedLaunchDarkly is a feature management platform that enables feature flagging and experimentation. It allows developers to control feature releases, experiment with features, and manage configuration settings in real-time.

## HashiCorp Vault

* HashiCorp Vault is a tool for managing secrets and protecting sensitive data. It provides a secure and centralized way to store and access credentials, API keys, and other secrets.

## Redis Cache:

* A computer code with colorful text

  Description automatically generated with medium confidenceRedis is an open-source, in-memory data structure store that can be used as a cache. Redis Cache provides fast and scalable caching for applications, improving performance by storing frequently accessed data in memory.

## Differences:

* These tools serve different purposes but can complement each other in building robust and scalable applications. **Serilog** enhances logging capabilities, **LaunchDarkly** enables feature management**, HashiCorp Vault** secures sensitive information, and **Redis Cache** boosts application performance through caching.

## CI/CD Implementation for Azure DevOps

* Continuous Integration (CI) and Continuous Delivery (CD) in Azure involve automating the software development process, from code changes to deployment. Azure DevOps Services is a comprehensive platform that facilitates CI/CD workflows.

### Continuous Integration (CI):

* + CI involves automatically building and testing code changes whenever developers commit to a version control repository. It helps identify integration issues early in the development process.

### Continuous Delivery (CD):

* + CD extends CI by automatically deploying code changes to a staging environment for further testing. It ensures that the application is always in a deployable state.

### Azure DevOps Services:

* + Azure DevOps provides a suite of services for CI/CD, including Azure Pipelines for building, testing, and deploying applications. It integrates with Azure Repos for version control and Azure Artifacts for package management.
  + Sample (Azure DevOps UI):
    - Set up pipelines using the Azure Pipelines visual designer, configuring build and release stages, triggers, and environments.
  + By implementing CI/CD with Azure, development teams can achieve faster release cycles, ensure code quality, and deliver software more reliably. Azure Pipelines, along with other Azure services, provides a seamless and customizable CI/CD experience.

## Azure Cloud

* Azure Cloud encompasses a vast array of services and features. Here are some key concepts, keywords, methods, and functions associated with Azure:

### Azure Resource Manager (ARM):

* + Explanation: ARM is the deployment and management service for Azure. It enables you to deploy and manage resources in an organized manner, using declarative templates.

### Azure CLI:

* + Explanation: Azure Command-Line Interface is a set of commands used to manage Azure resources. It provides a scriptable way to automate Azure tasks.

### Azure PowerShell:

* + Explanation: Azure PowerShell is a module that provides cmdlets to manage Azure resources using PowerShell scripting.

### Azure Portal:

* + Explanation: Azure Portal is a web-based interface for managing Azure resources. It provides a graphical way to interact with Azure services.

### Azure Functions:

* + Explanation: Azure Functions is a serverless compute service that allows you to run event-triggered code without explicitly provisioning or managing infrastructure.

### Azure Storage:

* + Explanation: Azure Storage is a scalable cloud storage solution. It includes services like Blob Storage, Table Storage, Queue Storage, and File Storage.

### Azure Virtual Machines (VMs):

* + Explanation: Azure VMs allow you to run Windows or Linux virtual machines in the cloud.
  + Sample: Create a VM using the Azure Portal or Azure CLI.

### Azure App Service:

* + Explanation: Azure App Service is a fully managed platform for building, deploying, and scaling web apps.
  + Sample: Deploy a .NET Core web app using Azure App Service.

### Azure Cognitive Services:

* + Explanation: Azure Cognitive Services provide AI and machine learning capabilities, including vision, speech, language, and decision services.
  + Sample: Use the Computer Vision API to analyze an image.

### Azure DevOps:

* + Explanation: Azure DevOps is a set of development tools that includes services for version control, build automation, release management, and more.
  + Sample: Create a build pipeline for a .NET application.

## Glossary for some Words

### As

* + As is like a type of chameleon, enabling safe type conversions or pattern matching in C#. It ensures compatibility and flexibility in working with different types.

### Base

* + Base is like the ancestral link, referencing the immediate parent class in an inheritance hierarchy. It provides access to the members of the base class.

### Case

* + Case is like a branching point, defining specific conditions within a switch statement. It directs the flow of execution based on the value of an expression.

### Catch

* + Catch is like a safety net, capturing and handling exceptions during runtime. It allows for graceful recovery from unexpected errors.

### Checked

* + Checked is like a guardian, ensuring that arithmetic overflow or underflow is detected and handled. It provides a safety net for numerical operations.

### Default

* + Default is like a fallback, providing the default value for a data type. It is used in scenarios where an explicit value is not assigned.

### Explicit

* + Explicit is like a clear instruction, specifying a clear and unambiguous conversion or implementation. It leaves no room for ambiguity in the compiler's decision-making.

### Extern

* + Extern is like an external liaison, indicating that the method is implemented outside the current codebase, often in native or external code.

### False

* + False is like a logical illusion buster, representing the Boolean value of false. It is a fundamental truth value in logical operations.

### Finally

* + Finally, is like a code custodian, specifying a block of code that will be executed whether an exception is thrown or not. It ensures cleanup operations.

### Fixed

* + Fixed is like an anchor, ensuring that the memory location of a variable remains constant. It's used in specific scenarios to work with pointers in a controlled manner.

### Implicit

* + Implicit is like an automatic bridge builder, allowing the compiler to automatically perform a conversion without explicit instruction. It simplifies type conversions in certain scenarios.

### In

* + In is like a data carrier, indicating that a parameter is passed by reference but is read-only within the method. It enhances performance while preserving immutability.

### Is

* + Is - is like a type of inspector, checking whether an object is of a specified type. It's used in type-checking scenarios.

### Lock

* + Lock is like a synchronization sentinel, ensuring exclusive access to a shared resource in a multithreaded environment. It prevents concurrent access conflicts.

### Namespace

* + Namespace is like a code organizer, providing a hierarchical structure for organizing and categorizing types. It helps prevent naming conflicts.

### New

* + New is like a code birth announcement, creating an instance of a class or invoking a constructor. It allocates memory for the new object.
  + The new constraint specifies that a type argument in a generic class or method declaration must have a public parameterless constructor. To use the new constraint, the type cannot be abstract.

### Null

* + Null is like an emptiness placeholder, representing the absence of a value or a reference that doesn't point to any object. It's a common sentinel value.

### Out

* + Out is like a data conveyor, indicating that a parameter is passed by reference and is an output parameter. It enables the method to return multiple values.

### Override

* + Override is like a method redefiner, indicating that a method in a derived class replaces a method in the base class. It enables polymorphic behavior.

### params

* + Params is like a variable parameter gatherer, allowing a variable number of parameters to be passed to a method. It simplifies method calls with varying argument counts.

### ref

* + Ref is like a reference bridge, indicating that a parameter is passed by reference. It enables the method to modify the original value.

### return

* + Return is like a result dispatcher, specifying the value to be returned from a method. It concludes the method's execution.

### sizeof

* + Sizeof is like a memory investigator, determining the size, in bytes, of a specified type. It's often used in low-level programming.

### stackalloc

* + Stackalloc is like a dynamic memory sculptor, allocating memory on the stack for arrays. It's suitable for short-lived, performance-critical scenarios.

### Struct

* + Struct is like a value-packed miniature, representing a lightweight, value-type class. It's suitable for small, frequently used data structures.

### this

* + This is like a self-reference pointer, referring to the current instance of the class. It disambiguates between instance and local variables.

### throw

* + Throw is like an exception catapult, signaling the occurrence of an exceptional condition. It interrupts the normal flow of execution.

### True

* + True is like a logical affirmation, representing the Boolean value of true. It's a fundamental truth value in logical operations.

### Try

* + Try is like a risk manager, encapsulating a block of code that might throw exceptions. It allows for controlled exception handling.

### typeof

* + typeof is like a type identifier, obtaining the Type object for a specified type. It's often used for reflection and dynamic operations.

### Unchecked

* + Unchecked is like a risk taker's flag, indicating that arithmetic overflow or underflow should be ignored. It allows unchecked numeric operations.

### Unsafe

* + Unsafe is like a warning sign, indicating that the code contains potentially unsafe operations. It's often used in low-level, performance-critical scenarios.

### Using

* + Using is like a resource custodian, ensuring proper disposal of resources after use. It simplifies resource management and prevents leaks.

### Virtual

* + Virtual is like an invitation to override, indicating that a method or property can be overridden in derived classes. It facilitates polymorphic behavior.

### Void

* + Void is like a non-returning ambassador, indicating that a method does not return a value. It's often used for methods with side effects.

### Volatile

* + Volatile is like a data freshness flag, indicating that a field may be changed by multiple threads. It ensures visibility of the latest value across threads.
* @, the verbatim identifier character.
* $, the interpolated string character.
* With <out T>, you can treat the interface reference as one upwards in the hierarchy.
* With <in T>, you can treat the interface reference as one downwards in the hiearchy.