**Explain the difference between .Net and C#**

.NET:

* Framework: .NET is a software development framework developed by Microsoft. It provides a large library of pre-coded solutions to common programming problems.
* Platform: .NET is a platform that supports several programming languages, including C#, F#, and Visual Basic.NET.
* Runtime: .NET includes the Common Language Runtime (CLR), which manages the execution of .NET programs and provides features like garbage collection and exception handling.
* Libraries: .NET provides a vast set of libraries and frameworks for building various types of applications, including web, desktop, mobile, and cloud applications.

C#:

* Language: C# (pronounced "C sharp") is a programming language designed by Microsoft for the .NET platform. It is a modern, object-oriented language with similarities to C++ and Java.
* Syntax: C# syntax is similar to other C-style languages and is designed to be simple, expressive, and easy to learn.
* Features: C# includes features such as strong typing, automatic memory management (garbage collection), and support for object-oriented programming concepts like classes, inheritance, and interfaces.
* Usage: C# is commonly used for developing Windows applications, web applications, and games, especially when targeting the .NET framework.

In summary, .NET is a framework/platform for building applications, while C# is a programming language used within the .NET ecosystem. C# is one of several languages that can be used to write .NET applications, but it is the most widely used and supported language in the .NET community.

**Confusion About .NET vs .NET Core vs .NET Standard vs .NET Framework.**

**.NET Standard:** This is a specification that defines a common set of APIs that can be used by .NET applications. This makes it possible to write code that can run on different .NET platforms, such as .NET Framework, .NET Core, and .NET 5 and beyond.

**.NET Framework:** This is the original version of the .NET platform, which is mostly used for building Windows desktops and web applications. It is still supported, but it is considered a legacy technology.

**.NET Core:** This is a cross-platform version of .NET that was introduced to overcome some limitations of the .NET Framework. It is lightweight, fast, and suitable for building cross-platform applications that can run on Windows, macOS, and Linux. .NET Core is the recommended choice for new projects.

**.NET 5 and beyond** Microsoft decided to simplify the naming and converge .NET Core and .NET Framework into a single platform called .NET 5. This new platform is fully supported and is the recommended choice for new projects.

In summary, if you are starting a new project, you should use the latest version of .NET Standard, such as .NET Standard 2.1 or later. This will give you the most modern features and improvements, and it will also make your application more cross-platform.

**Another answer**   
  
.NET is a cross language set of compilers and tools that are supported by an underlying set of runtime libraries. C# is the major .NET language, but also VB.NET and F# are also .NET languages.

**.NET Framework** is Windows only. The final version (v4.8.x) it is not actively being developed, but that doesn't mean it won't get security updates and it will be supported for as long as Window 11 is supported (and may be later OSs). <https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-framework>

.**NET Core** is supported on multiple operating systems (including Windows, Linux, macOs, IOS, Android OS and others). It was completely rewritten from scratch and although it shares a large compatibility with .NET framework, it is very different in many places. <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet>

.**NET 5 and above** is just .NET core version 5 and above. 5 was chosen to be higher than both .NET core 3, and .NET framework 4.8. (there was no .NET core 4.X)

.**NET Standard** is an in-between library that allows code to be shared between .NET Framework and .NET Core. In theory, if you compile code to .NET standard, it should be compatible with Framework and Core.

If you are looking for advice for which version of .NET to write code for (not really allowed on SO), it would generally be .NET core, unless you require one of the many Windows only APIs that are only available in .NET Framwork (although things like WCF is now on .NET core).

A screenshot of a computer screen

Description automatically generated

**What’s IL (Intermediate language) Code?**

IL (Intermediate Language) code, also known as MSIL (Microsoft Intermediate Language) or CIL (Common Intermediate Language), is the low-level, platform-independent binary instruction set used by the .NET Framework and .NET Core.

When you compile a .NET application, the source code is first compiled into IL code by the language compiler (such as C# or VB.NET). This IL code is then stored in an assembly along with metadata describing the types, members, and references used in the code.

IL code is not directly executable by the CPU. Instead, it is translated into native machine code by the JIT (Just-In-Time) compiler when the application is run. This allows .NET applications to be cross-platform, as the IL code can be run on any platform that has a compatible runtime environment.

IL code is human-readable and is often used for debugging purposes. Tools like IL Disassemblers can convert IL code back into a high-level language representation, making it easier to understand the behavior of .NET applications.

**What’s the use of JIT (Just in time compiler)?**

The Just-In-Time (JIT) compiler is a key component of the .NET runtime that plays a crucial role in executing .NET applications. Here's how it works and why it's important:

* **Dynamic Compilation**: Unlike languages like C or C++ that compile code into native machine code before execution, .NET languages (like C#) compile code into an intermediate language (IL). The IL code is then compiled into native machine code by the JIT compiler at runtime.
* **Optimization**: The JIT compiler can apply optimizations specific to the runtime environment and the underlying hardware. This means that the compiled code can be highly optimized for performance, taking advantage of the capabilities of the CPU and other hardware.
* **Cross-Platform Execution**: Because the IL code is compiled into native code at runtime, .NET applications can run on any platform that has a compatible JIT compiler. This allows .NET applications to be cross-platform, running on Windows, macOS, Linux, and other platforms supported by .NET runtimes like .NET Core and .NET 5+.
* **Execution Performance**: The JIT compiler can optimize code based on runtime information, such as the actual types of objects used in the code (known as "runtime type information" or RTTI). This can lead to better performance compared to statically compiled code in some cases.
* **Security**: The JIT compiler can also perform certain security checks and optimizations, such as array bounds checking and null reference checks, to help prevent certain types of runtime errors and security vulnerabilities.

In summary, the JIT compiler is a critical component of the .NET runtime that dynamically compiles IL code into native machine code at runtime, providing performance, cross-platform compatibility, and other benefits to .NET applications.

**- Decompile .NET DLL using Decompile or Decompiler (IL Spy or dotPeek).  
- Obfuscation**

**What’s CLR (Common Language Runtime)?**

What does the interviewer want to hear?

* CLR invokes JIT to compile to IL code.
* Cleans any unused objects by using GC.

The CLR (Common Language Runtime) is the execution engine that sits at the heart of the .NET Framework and its successor, .NET Core. It provides a range of services that allow developers to build and run applications written in various languages targeting the .NET platform. Here's a brief overview of the key features and functions of the CLR:

**Managed Execution**: The CLR provides a managed execution environment where applications are executed in a controlled manner. It manages memory allocation and deallocation, as well as thread execution and synchronization.

**Just-In-Time (JIT) Compilation**: The CLR uses a JIT compiler to convert Intermediate Language (IL) code into native machine code at runtime. This allows .NET applications to be platform-independent and provides performance optimizations tailored to the execution environment.

**Automatic Memory Management**: The CLR includes a garbage collector that automatically manages the allocation and deallocation of memory. This helps prevent memory leaks and makes memory management easier for developers.

**Exception Handling**: The CLR provides a robust mechanism for handling exceptions that occur during program execution. It allows developers to write code that gracefully handles errors and maintains the stability of the application.

**Security**: The CLR includes a security model that helps protect applications from malicious code and unauthorized access. It provides features such as code access security and role-based security to control access to resources.

**Language Interoperability**: The CLR supports multiple programming languages, allowing developers to write applications in the language of their choice. It provides a common runtime environment that enables interoperability between different languages.

**Base Class Library (BCL)**: The CLR includes a set of class libraries known as the Base Class Library (BCL). These libraries provide a wide range of functionality for tasks such as file I/O, networking, and data access, allowing developers to build applications more quickly and easily.

Overall, the CLR is a critical component of the .NET platform, providing the foundation for building and running a wide range of applications, from desktop applications to web services and beyond.

**What is managed and unmanaged code?**

A screenshot of a computer

Description automatically generated

Managed code refers to code that is executed by the Common Language Runtime (CLR) in the .NET Framework or .NET Core environment. The CLR provides services such as memory management, exception handling, and security for managed code. Languages like C#, Visual Basic .NET, and F# compile to managed code.

Unmanaged code, on the other hand, is code that is directly executed by the operating system without the assistance of a runtime environment like the CLR. It is typically written in languages like C or C++ and is responsible for managing its own memory, exceptions, and other low-level operations.

The key differences between managed and unmanaged code are:

1. **Memory Management**: Managed code relies on the CLR's garbage collector for memory management, while unmanaged code must handle memory allocation and deallocation manually.
2. **Exception Handling**: Managed code uses the CLR's exception handling mechanism, which provides a more structured approach to handling errors. Unmanaged code uses language-specific error handling mechanisms, such as C++'s try-catch blocks.
3. **Security**: Managed code benefits from the CLR's security features, such as code access security and role-based security. Unmanaged code must implement its own security measures.
4. **Performance**: Managed code can have performance overhead due to the CLR's runtime services. Unmanaged code can be more efficient in terms of performance but requires more careful management of resources.

Overall, managed code provides a higher level of abstraction and productivity for developers, while unmanaged code offers more control and performance optimization but requires more manual effort and attention to detail.

**Explain the importance of Garbage collector?**

A white background with black text

Description automatically generated

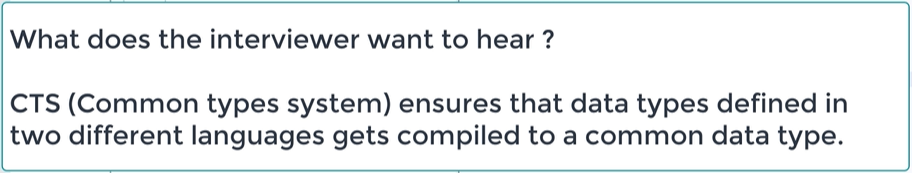
The Garbage Collector (GC) is a critical component of the .NET runtime environment, providing automatic memory management for applications. Here are some key reasons why the GC is important:

1. **Memory Management**: The GC automatically allocates and deallocates memory for objects, preventing memory leaks and simplifying memory management for developers. This helps avoid common programming errors related to manual memory management.
2. **Efficiency**: By reclaiming memory that is no longer in use, the GC helps optimize memory usage and improve application performance. It reduces the risk of memory fragmentation and improves the overall efficiency of memory allocation.
3. **Reliability**: Manual memory management can lead to bugs such as accessing deallocated memory (dangling pointers) or forgetting to deallocate memory (memory leaks). The GC helps eliminate these issues, improving the reliability and stability of applications.
4. **Simplicity**: Developers can focus more on writing application logic rather than managing memory, leading to faster development cycles and fewer programming errors related to memory management.
5. **Scalability**: Automatic memory management provided by the GC makes it easier to scale applications, as developers do not need to manually optimize memory usage for different scenarios or environments.
6. **Security**: The GC can help prevent security vulnerabilities such as buffer overflows and use-after-free errors, which can be exploited by attackers to compromise the security of an application.

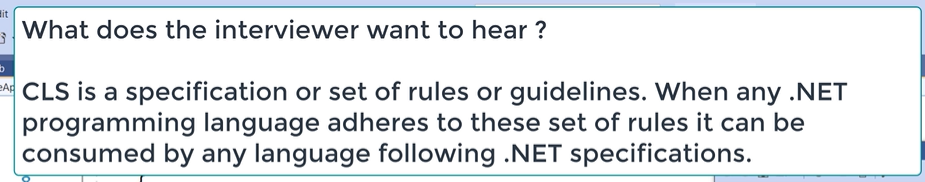
Overall, the GC plays a crucial role in modern programming languages like C# by providing automatic and efficient memory management, improving application reliability, performance, and security.

**(Using Key word - force Garbage collector – Performance Monitor)**[**https://www.youtube.com/watch?v=RgfuVp2lXIA&t=72s**](https://www.youtube.com/watch?v=RgfuVp2lXIA&t=72s)

**What is the importance of CTS?**



**Explain CLS?**



**Difference between stack vs Heap?**

In programming, "stack" and "heap" refer to two different areas of memory used for storing data. Here are the key differences between them:

1. **Memory Management**:
   * **Stack**: Memory on the stack is managed in a Last In, First Out (LIFO) manner. When a function is called, its local variables are pushed onto the stack, and when the function returns, those variables are popped off the stack.
   * **Heap**: Memory on the heap is managed more flexibly. Memory is allocated and deallocated manually by the programmer, typically using functions like **malloc** and **free** in C, or using language-specific memory management mechanisms like garbage collection in Java or C#.
2. **Access Speed**:
   * **Stack**: Accessing memory on the stack is generally faster than on the heap because the stack is a simple data structure with a fixed-size allocation.
   * **Heap**: Accessing memory on the heap is slower because the memory allocation and deallocation involve more complex operations.
3. **Memory Allocation**:
   * **Stack**: Memory on the stack is allocated in a contiguous block, and the size of this block is determined at compile time.
   * **Heap**: Memory on the heap is allocated in a more dynamic and flexible manner. The size of the memory block can be determined at runtime, and memory can be allocated and deallocated as needed.
4. **Lifetime**:
   * **Stack**: Variables on the stack have a limited lifetime. They are created when a function is called and destroyed when the function returns.
   * **Heap**: Memory on the heap can have a longer lifetime. It can persist beyond the scope of a single function and can be accessed from multiple parts of a program.
5. **Usage**:
   * **Stack**: The stack is typically used for storing function parameters, return addresses, and local variables.
   * **Heap**: The heap is used for dynamically allocated memory, such as when you need to allocate memory for an array whose size is not known until runtime or for objects in object-oriented programming languages.

In summary, the stack is fast and has a limited lifetime, making it suitable for storing temporary data, while the heap is more flexible and can be used for storing data with a longer lifetime or data whose size is not known at compile time.

**What are value types and Reference types?**

The key difference between value types and reference types is how they are stored and accessed in memory. Value types are typically stored on the stack, and their value is directly accessed. Reference types, on the other hand, are stored on the heap, and variables hold references to their location in memory.

It's important to understand these differences, as they can affect how you work with variables and objects in your code, especially when it comes to passing them as arguments to methods or assigning them to other variables.

**The Key Difference**

The key difference between value types and reference types is that value types hold the actual data, while reference types hold a reference to the data's location in memory. This fundamental difference leads to different behavior in the way these types are handled in the code, and it's essential to understand this to avoid unintended side effects.

Whether to use value types or reference types depends on the specific needs of your program. Value types are generally more efficient in terms of memory and performance, but they lack the flexibility and functionality provided by reference types. Conversely, reference types are more flexible and powerful, but they can lead to potential pitfalls if not handled correctly, like unexpected side effects from changing data in one place that another part of your code is also referencing.

**What's boxing and unboxing in c#?**

Boxing and unboxing are concepts in C# (and other languages like Java) related to the conversion between value types and reference types.

* **Boxing**: When a value type (e.g., int, double, struct) is converted to an object type (e.g., object, interface), it is called boxing. The value is placed in a box (an instance of the object type), and this box is stored on the heap. Boxing allows value types to be treated as objects.
* **Unboxing**: When an object that was previously boxed is converted back to its original value type, it is called unboxing. Unboxing involves extracting the value from the boxed object. Unboxing requires an explicit cast and is a way to access the value stored in the object.

Here's an example:

int i = 42; // Value type

object obj = i; // Boxing: converting int to object

int j = (int)obj; // Unboxing: converting object back to int

Boxing and unboxing can incur a performance overhead because of the conversion process. It's generally more efficient to use value types directly when possible, rather than boxing and unboxing them.

**Explain casting, implicit casting and explicit casting?**

Casting is the process of converting a value from one data type to another. There are two types of casting: implicit casting and explicit casting.

1. **Implicit Casting**: Implicit casting occurs when the destination data type can fully represent all possible values of the source data type without losing any information. This type of casting is done automatically by the compiler.

int numInt = 10;

double numDouble = numInt; // Implicit casting from int to double

1. **Explicit Casting**: Explicit casting, also known as type casting, is used when you want to convert a value from a larger data type to a smaller data type, or when the destination data type cannot fully represent all possible values of the source data type. Explicit casting requires the use of a cast operator.

double numDouble = 10.5;

int numInt = (int)numDouble; // Explicit casting from double to int

Explicit casting can result in a loss of data if the value being casted cannot be fully represented in the destination data type. It's important to be cautious when using explicit casting to avoid loss of precision or data.

In C#, you can use explicit casting to convert between numeric types, as well as between other compatible types such as enums and their underlying types, and between objects and interfaces that they implement.

**Array vs ArrayList?**

A white rectangular sign with black text

Description automatically generated

Certainly! Here's a summary of the differences between Array and ArrayList:

1. **Type**:
   * **Array**: Fixed size, specific type.
   * **ArrayList**: Dynamic size, holds objects of any type.
2. **Size**:
   * **Array**: Size is fixed and specified at creation.
   * **ArrayList**: Size can change dynamically.
3. **Performance**:
   * **Array**: Generally more performant due to fixed size.
   * **ArrayList**: Slight performance overhead due to dynamic resizing.
4. **Type Safety**:
   * **Array**: Type-safe, enforced at compile time.
   * **ArrayList**: Type safety enforced at runtime.
5. **Usage**:
   * **Array**: Use when size is known and performance is critical.
   * **ArrayList**: Use when size is dynamic and type flexibility is needed.

In modern C#, consider using generic collections like **List<T>** instead of ArrayList for better type safety and performance.

**What are generic collections?Top of Form**

A screenshot of a computer

Description automatically generated

**What are Threads?**

Threads are the smallest unit of execution within an operating system's scheduling system. They are independent sequences of execution within a process. A process can have multiple threads running concurrently, allowing it to perform multiple tasks at the same time or to handle multiple operations asynchronously.

Threads share the same memory space within a process, which allows them to access the same data. This can make concurrent programming complex, as developers need to ensure that threads do not interfere with each other's operations (known as thread safety). However, threads can also communicate with each other through shared memory, which can be useful for coordinating tasks or passing data between threads.

Threads are used in many types of applications, including those that require responsiveness, parallel processing, or multitasking. They are commonly used in graphical user interfaces (GUIs), web servers, and applications that perform background tasks.

**Thread vs Task?**

Threads and tasks are both mechanisms used for concurrent programming, but they have different characteristics and are used in different programming paradigms:

1. **Threads**:
   * Threads are the smallest unit of execution managed by the operating system.
   * They are heavy-weight, requiring significant system resources (such as memory and CPU time) to create and manage.
   * Threads have a one-to-one mapping with operating system threads, meaning each thread corresponds to a separate system-level thread.
   * Threads are managed explicitly by the programmer, requiring careful synchronization and coordination to avoid issues like race conditions and deadlocks.
2. **Tasks**:
   * Tasks are a higher-level abstraction provided by many modern programming frameworks, including .NET's Task Parallel Library (TPL) and Java's Executor framework.
   * Tasks are lightweight and are not directly mapped to operating system threads. Instead, they are managed by a task scheduler that maps them to a pool of threads.
   * Tasks can represent asynchronous operations or parallelizable units of work. They can be used to improve the responsiveness and scalability of applications.
   * Tasks provide higher-level constructs for working with concurrency, such as continuations, cancellation, and error handling, making concurrent programming easier and safer.

In summary, threads are a lower-level construct provided by the operating system, while tasks are a higher-level abstraction provided by programming frameworks. Tasks are often preferred for concurrent programming in modern applications due to their lightweight nature and higher-level abstractions.

**What’s a delegate?**

A screenshot of a computer

Description automatically generated

**Delegates Overview**

Delegates have the following properties:

* Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.
* Delegates allow methods to be passed as parameters.
* Delegates can be used to define callback methods.
* Delegates can be chained together; for example, multiple methods can be called on a single event. (Multicast)
* Methods don't have to match the delegate type exactly. For more information, see [Using Variance in Delegates](https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/using-variance-in-delegates).
* Lambda expressions are a more concise way of writing inline code blocks. Lambda expressions (in certain contexts) are compiled to delegate types. For more information about lambda expressions, see [Lambda expressions](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions).

**Delegate vs Events**

// Events uses Delegates

// Delegates are for callbacks , not encapsulated

// Events publisher subscriber model , encapsulated

In summary, delegates are used to create method pointers, while events are a higher-level concept built on delegates, used for implementing the observer pattern in C#.

**What's Object-Oriented Programming (OOP)?**

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data in the form of fields (attributes or properties) and code in the form of procedures (methods or functions).

**What’s Functional programming?**

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. Key concepts in functional programming include:

1. **First-class functions**: Functions are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.
2. **Pure functions**: Functions that always return the same result for the same input and have no side effects (i.e., they do not modify global state or have observable interactions with the outside world).
3. **Immutable data**: Data is treated as immutable, meaning once it is created, it cannot be changed. Instead, new values are created based on existing ones.
4. **Higher-order functions**: Functions that can take other functions as arguments or return functions as results. They enable functions to be more reusable and composable.
5. **Recursion**: A technique where a function calls itself to solve smaller instances of the same problem, often replacing loops in imperative programming.

Functional programming languages, such as Haskell, Lisp, and Erlang, emphasize the use of these concepts to write concise, predictable, and easier-to-reason-about code. While many languages support some functional programming features, pure functional languages adhere strictly to these principles.

**What's abstraction?**

Abstraction is a fundamental concept in programming that involves hiding the complex implementation details of a system or an object and only showing the necessary features or functionality. It allows developers to work with high-level concepts without needing to understand the low-level details.

In object-oriented programming (OOP), abstraction is often achieved through the use of abstract classes and interfaces. Abstract classes define common characteristics of subclasses but cannot be instantiated themselves. Interfaces define a contract that classes can implement, specifying methods that must be implemented but not providing the implementation details.

Abstraction helps in managing complexity, improving code readability, and promoting code reuse. By abstracting away unnecessary details, developers can focus on the essential aspects of a problem, leading to more maintainable and flexible code.

**What's encapsulation?**

Encapsulation is the concept of bundling the data (variables) and methods (functions) that operate on the data into a single unit or class. It hides the internal state of an object from the outside world and only exposes a controlled interface for interacting with the object. This helps in controlling access to the data, ensuring that it is accessed and modified in a consistent and secure manner.

Encapsulation is a key principle of object-oriented programming (OOP) and is achieved by using access modifiers such as private, protected, and public in languages like C# and Java. Private members can only be accessed within the same class, while protected members can be accessed by subclasses as well. Public members are accessible from outside the class.

Encapsulation helps in achieving data abstraction, as the internal representation of an object is hidden from the outside world. It also helps in maintaining the integrity of the data by preventing direct access to it, allowing only controlled access through methods.

**Difference between abstraction and encapsulation?**

**Abstraction is generalized term. i.e. Encapsulation is subset of Abstraction.**

A screenshot of a computer

Description automatically generated

Example: The **solution architect** is the person who creates the high-level **abstract** technical design of the entire solution, and this design is then handed over to the the **development team** for **implementation**.  
Here, solution architect acts as a abstract and development team acts as a Encapsulation.

**What's inheritance?**

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (derived class or subclass) to inherit properties and behavior from an existing class (base class or superclass). The derived class can extend or override the functionality of the base class, while still inheriting its attributes and methods.

Inheritance enables code reuse and promotes the creation of a hierarchy of classes that represent different levels of abstraction. It allows you to define a general class with common attributes and behaviors, and then create more specific classes that inherit from it and add their own specialized features.

For example, you could have a base class called **Shape** with properties like **Color** and methods like **Draw**, and then create derived classes like **Circle** and **Rectangle** that inherit from **Shape** and add specific properties and methods related to circles and rectangles.

Inheritance is often used to model real-world relationships and hierarchies, making code more organized, maintainable, and scalable. However, it should be used judiciously, as excessive use of inheritance can lead to complex and tightly coupled class hierarchies, making the code harder to understand and maintain.

**What’s Polymorphism**

Polymorphism in OOPs is one of the four important principles of object-oriented programming (OOP), which allows objects of different classes, objects, variables and methods to exist in different forms. With the help of polymorphism in oops, it is easy to represent one single form into various forms.

There are two main types of polymorphism in object-oriented programming. These are compile-time polymorphism and Run-time polymorphism. Polymorphism allows flexibility and increases efficiency while designing software systems. We can add new classes easily with the help of polymorphism without modifying existing entities.

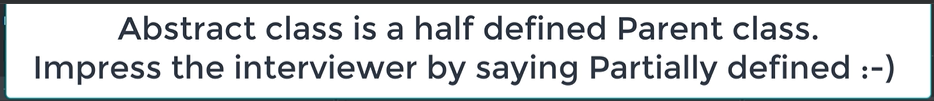
There are two main types of polymorphism:

1. **Compile-time (or static) polymorphism:** This is achieved using method overloading and operator overloading. Method overloading allows multiple methods with the same name but different parameters to coexist in the same class, while operator overloading allows operators to be redefined for custom classes. (like + operator, we can make it perform another job)
2. **Run-time (or dynamic) polymorphism:** This is achieved using method overriding. Method overriding occurs when a subclass provides a specific implementation of a method that is already provided by its superclass. When an overridden method is called through a superclass reference, the method of the subclass is executed.

Polymorphism is a powerful concept that allows for more flexible and modular code, as it enables you to write code that can work with objects of different types without needing to know their specific classes.

**What**

**‘s abstract class?**



An abstract class in C# is a class that cannot be instantiated directly. It is designed to be a base class for other classes to inherit from. Abstract classes can contain abstract methods, which are methods without a body, and concrete methods, which have a body.

Abstract classes are used when you have a base class that defines some common behavior, but you want to leave the implementation of certain methods to the derived classes. This allows you to create a common interface for a group of related classes, while still allowing each class to provide its own implementation of certain methods.

**Is an abstract method virtual?**

In C#, abstract methods are implicitly virtual, meaning they can be overridden in derived classes. However, the **virtual** keyword is not used when declaring an abstract method.

**What's an interface?**

An interface in C# is a reference type that defines a contract for classes to implement. It contains only the signatures of methods, properties, events, or indexers. An interface does not provide implementation for any members; it only declares them.

A white text with black text

Description automatically generated

**Abstract class vs Interface**

A white rectangular box with black text

Description automatically generated

A white background with black text

Description automatically generated

A blue background with yellow text

Description automatically generated

Interfaces and abstract classes are both used to define contracts for classes, but they have some key differences:

1. **Definition**:
   * An interface defines only method signatures, properties, events, and indexers but not their implementation.
   * An abstract class can have both abstract members (methods, properties, etc., without implementation) and concrete members (with implementation).
2. **Multiple Inheritance**:
   * A class can implement multiple interfaces, allowing for multiple inheritance of contracts.
   * A class can inherit from only one abstract class, restricting it to a single base class.
3. **Implementation**:
   * Interfaces cannot contain implementation. Any class that implements an interface must provide its own implementation for the members.
   * Abstract classes can provide default implementation for some members, which can be optionally overridden by derived classes.
4. **Access Modifiers**:
   * Interface members are by default public and cannot have access modifiers.
   * Abstract class members can have access modifiers like public, protected, internal, etc.
5. **Construction**:
   * Interfaces cannot have constructors.
   * Abstract classes can have constructors, which are used for initialization when a concrete class is instantiated.
6. **Usage**:
   * Use interfaces when you want to define a contract that can be implemented by unrelated classes to provide a common behavior.
   * Use abstract classes when you want to provide a partial implementation of a class and leave the rest to be implemented by derived classes.

In summary, use interfaces for defining contracts and achieving polymorphism across unrelated classes, and use abstract classes for providing a base implementation that can be shared among derived classes.

**What’s constructor**

A constructor in C# is a special type of method that is used to initialize objects. It has the same name as the class and does not have a return type, not even void. Constructors are called when an instance of a class is created using the **new** keyword. They are used to set initial values for object properties, perform any necessary setup, and ensure that the object is in a valid state after it is created.

A diagram of a child constructor

Description automatically generated

**Constructors - C# \_ Microsoft Learn.pdf**

**Static Constructors - C# \_ Microsoft Learn.pdf / Private Constructors - C# \_ Microsoft Learn.pdf**

More points that need more study time (Private Constructor, Static Constructor).

A white text with black text

Description automatically generated with medium confidence

A white text on a white background

Description automatically generated

A screenshot of a white text

Description automatically generated

**What's SOLID principles?**

SOLID is an acronym that represents five important principles of object-oriented programming and design. These principles help developers create software that is easier to maintain, understand, and extend over time.

**The Single Responsibility Principle (SRP)** states that a class should have only one reason to change, meaning it should have only one responsibility. This principle suggests that a class should do one thing and do it well. If a class has multiple responsibilities, it becomes harder to understand, maintain, and change over time.

**The Open/Closed Principle (OCP)** is a principle in object-oriented design that states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to extend the behavior of a system without modifying its existing code.

In practical terms, this means that when you want to add new functionality, you should do so by adding new code, not by changing existing code. This can be achieved through the use of interfaces, abstract classes, and design patterns like the Strategy pattern or the use of dependency injection.

By adhering to the OCP, you can ensure that your codebase remains stable and maintainable, as changes to one part of the system are less likely to have unintended consequences elsewhere.

**The Liskov Substitution Principle (LSP)** is a principle in object-oriented programming that states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In other words, a subclass should be able to substitute its superclass without changing the behavior of the program.

This principle is important for maintaining the correctness of the program when using inheritance. It ensures that subclasses adhere to the contract established by the superclass, allowing for polymorphic behavior without unexpected side effects.

For example, consider a class **Rectangle** with **Width** and **Height** properties and a method **SetWidth** to set the width of the rectangle. According to LSP, if we have a subclass **Square** that extends **Rectangle**, it should not override the **SetWidth** method to maintain the behavior of a **Rectangle**. Instead, it should inherit the behavior from **Rectangle** and maintain the LSP.

To adhere to the Liskov Substitution Principle, use the override keyword to extend or modify the behavior of base class methods in a way that is consistent with the base class's contract. Be cautious with the new keyword, as it can introduce hidden behaviors that break substitutability and lead to LSP violations.

**The Interface Segregation Principle (ISP)** states that a client should not be forced to depend on interfaces it does not use. This principle helps to ensure that classes remain focused and not overloaded with unnecessary dependencies.

In practical terms, ISP suggests that interfaces should be fine-grained and specific to the needs of the clients that use them. Instead of having large interfaces that encompass a wide range of methods, it's better to have smaller, more specialized interfaces that cater to specific functionalities.

**The Dependency Inversion Principle (DIP)**

The dependency inversion principle is very simple, and it states that high level parts of the system

should not depend on low level parts of the system directly, that instead they should depend on some

kind of abstraction.

**What are threads in C#**

In C#, a thread is the smallest unit of execution that can be scheduled by an operating system. Threads allow a program to perform multiple tasks concurrently or asynchronously. In a multithreaded application, multiple threads can execute independently, sharing the same memory space and resources. This allows for tasks to be performed concurrently, potentially improving performance and responsiveness of the application.

**What's indexer in C#**

In C#, an indexer is a special type of property that enables an object to be indexed like an array. It allows instances of a class or struct to be accessed using the same syntax as an array. Indexers are defined using the **this** keyword followed by one or more parameters, which act as the index. They are useful when you want to provide array-like access to the elements of a class or struct.

**What's mutable vs immutable in C#**

In C#, mutable and immutable refer to the ability to change the state of an object after it has been created.

* **Mutable:** Objects whose state can be changed after creation are mutable. For example, if you have a class with properties that can be modified, it's considered mutable.
* **Immutable:** Objects whose state cannot be changed after creation are immutable. Immutable objects are typically created with all their data set in the constructor and do not provide methods to modify their state. Once created, their state remains constant.

For example, strings in C# are immutable. Once a string is created, you cannot change its value directly. Any operation that appears to modify a string actually creates a new string object with the modified value. This immutability ensures that once you have a string, its value remains the same, which can be useful for certain scenarios like caching or thread safety.

A screenshot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**Static vs read only in C#**

In C#, **static** and **readonly** are both used to define fields, but they serve different purposes:

* **Static:** A **static** field is associated with the type itself rather than with any specific instance of the type. This means that all instances of the type share the same value for a static field. Static fields are initialized only once, typically when the type is first accessed, and they retain their value for the lifetime of the application domain. You can access a static field using the type name, e.g., **ClassName.StaticFieldName**.
* **ReadOnly:** A **readonly** field is one whose value can be assigned at initialization or within the constructor of the class where it is declared. Once assigned, the value of a **readonly** field cannot be changed. Each instance of the class can have a different value for a **readonly** field, but once set, that value cannot be modified. **readonly** fields are useful when you want to ensure that a field's value is set only once and remains constant thereafter.

In summary, **static** is used to create fields that are shared among all instances of a class and whose value remains constant throughout the application's lifetime, while **readonly** is used to create fields that are initialized once and cannot be changed thereafter for each instance of a class.

**Release vs Debug.**

In C#, "Release" and "Debug" are configurations used during the build process to specify different settings for compiling and running the application:

1. **Debug Configuration:**
   * This configuration is typically used during development and debugging.
   * Debug builds include additional debugging information, such as symbols, which makes it easier to debug the code.
   * Optimizations are usually disabled to provide better support for debugging, which may result in slower performance.
   * Debugging symbols allow developers to set breakpoints, step through code, inspect variables, and perform other debugging tasks more effectively.
2. **Release Configuration:**
   * This configuration is used for deployment in a production environment.
   * Release builds are optimized for performance and may have debugging information stripped out to reduce the size of the executable.
   * Compiler optimizations are enabled, resulting in faster execution speed and smaller code size.
   * Release builds do not include symbols, making it harder to debug the code in case of issues. Therefore, debugging in a release build is generally more challenging.

In summary, the "Debug" configuration is suited for development and debugging tasks, while the "Release" configuration is optimized for performance and is used for the final deployment of the application.

**What's Weak references?**

Weak references in C# are a way to reference an object without preventing it from being garbage collected. When you create a weak reference to an object, the garbage collector can still collect the object if there are no strong references to it, even if there are weak references. Weak references are useful in scenarios where you want to keep track of an object but don't want to prevent it from being collected when it's no longer needed.

**What's Yield in C#**

In C#, **yield** is a keyword used in iterator methods to simplify the process of creating custom iterators. When you use **yield** in a method, you're indicating that the method is an iterator, and it will return elements one at a time, rather than returning a collection all at once.

**Explain to me the encryption techniques.**

Encryption is a process of converting plaintext (human-readable data) into ciphertext (unreadable data) using an algorithm and a key. This ensures that unauthorized parties cannot read the data without the correct key to decrypt it. There are several encryption techniques, including:

1. **Symmetric Encryption**: In this technique, the same key is used for both encryption and decryption. Examples include AES (Advanced Encryption Standard) and DES (Data Encryption Standard).
2. **Asymmetric Encryption**: Also known as public-key encryption, this technique uses a pair of keys - a public key for encryption and a private key for decryption. Examples include RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography).
3. **Hashing**: While not technically encryption, hashing is a cryptographic technique used to convert data into a fixed-length string of bytes, called a hash value. Hashing is a one-way process, meaning that the original data cannot be easily obtained from the hash value. Examples include MD5 and SHA-256.
4. **Key Exchange**: This technique is used to securely exchange encryption keys between parties without the risk of interception. Examples include Diffie-Hellman key exchange and ECDH (Elliptic Curve Diffie-Hellman).
5. **Digital Signatures**: Digital signatures are used to verify the authenticity and integrity of a message or document. They are created using a private key and can be verified using the corresponding public key. Examples include RSA signatures and ECDSA (Elliptic Curve Digital Signature Algorithm).

Each encryption technique has its strengths and weaknesses, and the choice of technique depends on the specific requirements of the application, including security level, performance, and compatibility.

**Explain encoding?**

Encoding is the process of converting data from one format to another. In the context of text and characters, encoding refers to the representation of characters as bytes (or bits) for storage or transmission. This is necessary because computers store and process data in binary format, and text characters need to be converted to binary before they can be stored or transmitted.

There are many different encoding schemes, each with its own way of representing characters as bytes. Some common encoding schemes include:

1. **ASCII (American Standard Code for Information Interchange)**: ASCII is a 7-bit encoding scheme that represents 128 characters, including uppercase and lowercase letters, digits, punctuation marks, and control characters.
2. **Unicode**: Unicode is a universal character encoding standard that aims to represent every character from every language in the world. Unicode uses variable-length encoding, with the most common characters represented using 16 bits (UTF-16 encoding) or 8 bits (UTF-8 encoding).
3. **UTF-8**: UTF-8 is a variable-length encoding scheme for Unicode characters. It uses 8-bit code units and can represent the entire Unicode character set.
4. **UTF-16**: UTF-16 is another variable-length encoding scheme for Unicode characters. It uses 16-bit code units and can represent the entire Unicode character set.
5. **ISO-8859-1 (Latin-1)**: ISO-8859-1 is an 8-bit encoding scheme that represents the first 256 characters of the Unicode character set. It is commonly used for Western European languages.

When encoding text, it's important to use the correct encoding scheme to ensure that the data is interpreted correctly by the receiving system. Using the wrong encoding can result in data corruption or the display of incorrect characters.

**Explain base 64 encoding?**

Base64 encoding is a method of encoding binary data, such as images, audio, or other types of files, into a text format that is safe for transmission over text-based protocols, such as email or HTML. It works by converting 3 bytes of binary data into 4 printable ASCII characters.

Base64 encoding is commonly used for encoding binary data in email attachments, embedding images in web pages, and other similar purposes.