.NET framework and C# basics

Contents

.NET framework basics

What is .NET Framework?
The **.NET Framework** is a software framework developed by Microsoft. It provides a controlled environment for developing and running applications. It simplifies application development by offering services like memory management, security, and application deployment.

Languages in .NET Framework
The .NET Framework supports multiple languages, such as:

- **C#** (files with .cs extension)
- **VB.NET** (files with .vb extension)
- And many more...

What Happens When an Application is Compiled and Executed?

1. **Compilation**:
    - When a .NET program is compiled with the appropriate compiler, the output is an **assembly**.
    - Assemblies can either be in .dll (Dynamic Link Library) or .exe (Executable) format.
2. **Contents of an Assembly**:
    - Assemblies contain code in **Microsoft Intermediate Language (MSIL)** or **Common Intermediate Language (CIL)** format.
    - An assembly is essentially a collection of **MSIL code** and **metadata**.
3. **Purpose of .dll and .exe:**
    - **.dll**: Stands for **Dynamic Link Library** and is used as a library application in .NET.
    - **.exe**: Represents an executable application and contains a starting point of execution, such as the Main method.

Execution Process of Assemblies
To execute assemblies, the system must have the **Common Language Runtime (CLR)** installed. The CLR provides a **managed runtime infrastructure** for .NET applications.
Key Roles of CLR:

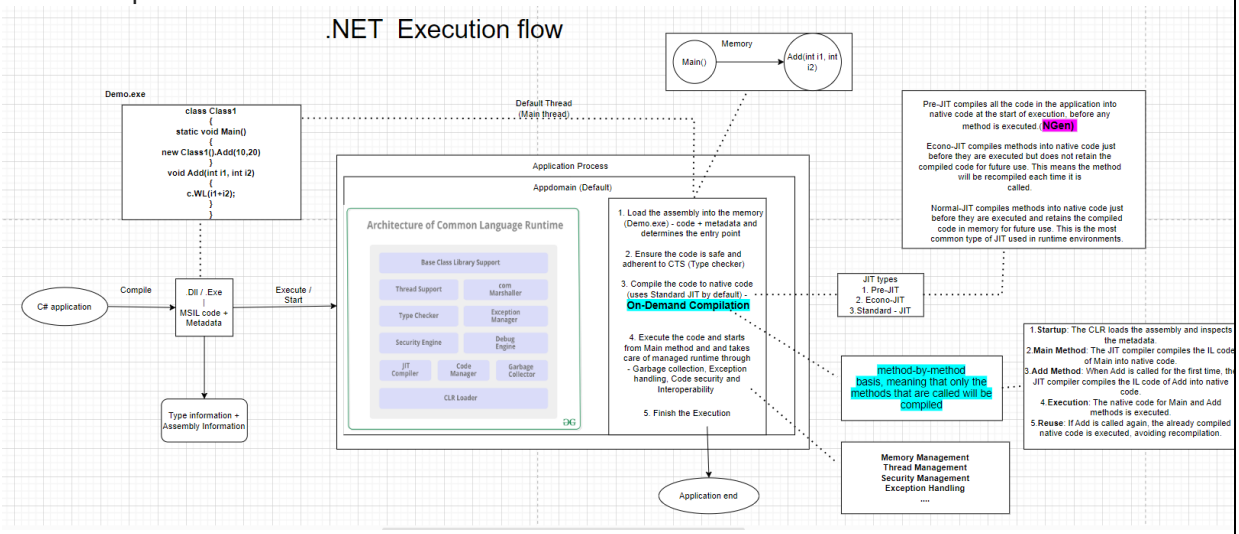
1. **JIT Compilation**: Converts MSIL code into native machine code through the **Just-In-Time (JIT) compiler**.
2. **Performance Optimization**: Manages memory and optimizes the performance of .NET applications.
3. **Code Verification**: Performs various code checks to ensure safety and reliability.

Flow of Execution in .NET Framework:
plaintext
Copy code
C# (.cs) VB.NET (.vb) .....
| Compile
|
Assembly (.exe/.dll) ---- (MSIL / CIL)
|
| CLR (JIT Compiler)

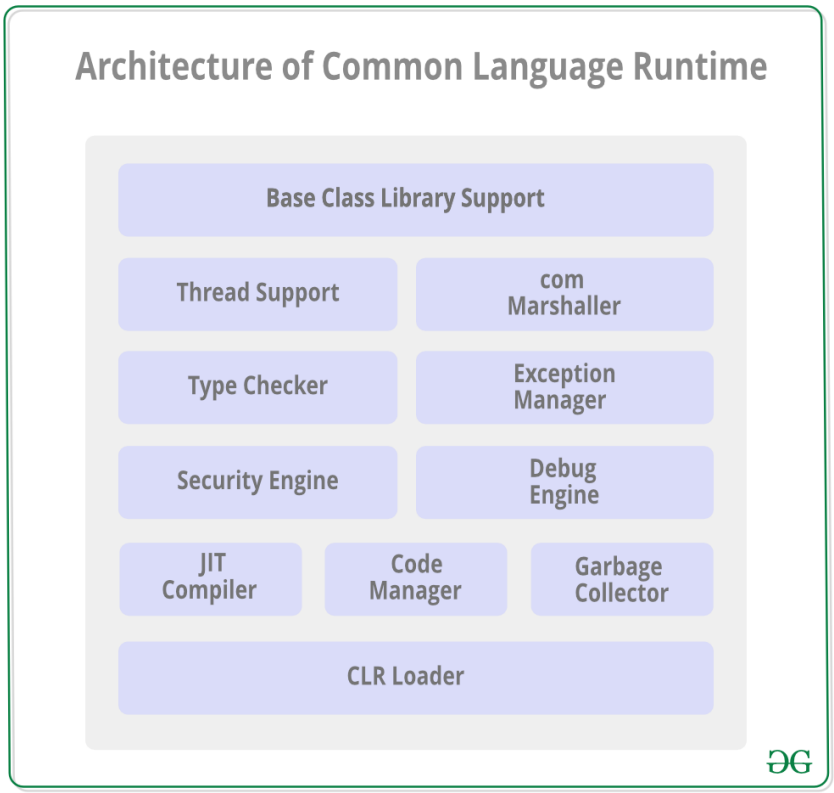
|
Native Code
|
Operating System (OS)

Common Language Runtime

The Common Language Runtime (CLR) in .NET provides a managed execution environment for .NET applications. The CLR handles various aspects of execution, including memory management, security, and exception handling. Here's a detailed breakdown of the CLR code execution process:



Architecture of Common Language Runtime



There are multiple components in the architecture of Common Language Runtime.

Base Class Library Support: The Common Language Runtime provides support for the base class library. The BCL contains multiple libraries that provide various features such as *Collections*, *I/O*, *XML*, *DataType definitions*, etc. for the multiple *.NET* programming languages.

Thread Support: The CLR provides thread support for managing the parallel execution of multiple threads. The *System.Threading class* is used as the base class for this.

COM Marshaller: Communication with the COM (Component Object Model) component in the .NET application is provided using the COM marshaller. This provides the COM interoperability support.

Type Checker: Type safety is provided by the type checker by using the Common Type System (CTS) and the Common Language Specification (CLS) that are provided in the CLR to verify the types that are used in an application.

Exception Manager: The exception manager in the CLR handles the exceptions regardless of the *.NET Language* that created them. For a particular application, the catch block of the exceptions are executed in case they occur and if there is no catch block then the application is terminated.

Security Engine: The security engine in the CLR handles the security permissions at various levels such as the code level, folder level, and machine level. This is done using the various tools that are provided in the *.NET* framework.
Debug Engine: An application can be debugged during the run-time using the debug engine. There are various ICorDebug interfaces that are used to track the managed code of the application that is being debugged.
JIT Compiler: The JIT compiler in the CLR converts the Microsoft Intermediate Language (MSIL) into the machine code that is specific to the computer environment that the JIT compiler runs on. The compiled MSIL is stored so that it is available for subsequent calls if required.
Code Manager: The code manager in CLR manages the code developed in the .NET framework i.e. the managed code. The managed code is converted to intermediate language by a language-specific compiler and then the intermediate language is converted into the machine code by the Just-In-Time (JIT) compiler.
Garbage Collector: Automatic memory management is made possible using the garbage collector in CLR. The garbage collector automatically releases the memory space after it is no longer required so that it can be reallocated.
CLR Loader: Various modules, resources, assemblies, etc. are loaded by the CLR loader. Also, this loader loads the modules on demand if they are actually required so that the program initialization time is faster and the resources consumed are lesser.

Steps in CLR Code Execution Process

1. Source Code Compilation
- When C# (or another .NET language) source code is compiled using a compiler (e.g., csc.exe for C#), it is translated into Microsoft Intermediate Language (MSIL or IL) and stored in an assembly (a .dll or .exe file).

2. Assembly Loading
- At runtime, the CLR loads the necessary assemblies into memory, reading the metadata and IL code from the assembly files.

3. Verification
- The CLR verifies the IL code to ensure it is safe for execution.
- This involves checking type safety and adherence to the Common Type System (CTS) rules.

4. Just-In-Time (JIT) Compilation

- **On-Demand Compilation:** The JIT compiler translates IL code into native machine code just before execution, on a method-by-method basis.
- **Optimizations:** Various performance improvements are applied during the translation.

5. Execution

- Native code is executed by the CPU. During execution, the CLR provides:
  - **Memory management** (via garbage collection)
  - **Exception handling**
  - **Security enforcement**

6. Garbage Collection

- The garbage collector (GC) automatically manages memory allocation and deallocation.
- Objects no longer in use are periodically removed to prevent memory leaks and optimize memory usage.

7. Exception Handling

- The CLR handles exceptions in a structured way using try, catch, and finally blocks, ensuring proper resource cleanup.

8. Security

- The CLR enforces security using mechanisms like Code Access Security (CAS) to ensure operations adhere to defined permissions and policies.

9. Interoperability

- The CLR supports interoperability with unmanaged code, allowing .NET applications to call native APIs or integrate with COM components.

Detailed Workflow

1. Application Start

- The operating system loader initializes the CLR within the application's process space.

2. Metadata Inspection

- The CLR inspects the assembly's metadata to identify the entry point (e.g., Main method in C#).

3. Class Loading

- Required classes are loaded, and static fields/constructors are initialized.

4. JIT Compilation and Execution

- The JIT compiler converts IL code of the entry method into native code just before execution.
- Subsequent methods are JIT-compiled as they are invoked.

5. Runtime Services

The CLR continuously provides:

- **Memory Management:** Handles allocation, deallocation, and garbage collection.
- **Thread Management:** Manages threads and synchronization.
- **Security Management:** Enforces security policies and permissions.
- **Exception Handling:** Ensures proper handling and cleanup of exceptions.

6. Application End

- The CLR performs cleanup tasks such as object finalization and resource release when the application terminates.

The Default Thread

1. Entry Point Execution
- The application's entry point (e.g., Main method) runs on the default main thread created by the OS.

2. Main Thread Characteristics
- **Console Applications:** The Main method runs on the main thread.
- **GUI Applications:** The main thread also handles the UI event loop, user inputs, and UI updates.

Types of JIT Compilation

1. Pre-JIT (Pre-JIT Compilation)
- **Description:** Compiles all IL code into native code before execution starts.
- **Usage:** Scenarios requiring fast runtime performance.
- **Pros:** Faster execution after initial compilation.
- **Cons:** Longer startup time.
- **Tool:** NGen (Native Image Generator).

2. Econo-JIT (Economy JIT)
- **Description:** Compiles methods just before execution but does not retain compiled code.
- **Usage:** Resource-constrained environments.
- **Pros:** Lower memory usage.
- **Cons:** Slower performance for frequently called methods.

3. Normal-JIT (Standard JIT)
- **Description:** Compiles methods just before execution and retains compiled code for future use.
- **Usage:** Default in general-purpose applications.
- **Pros:** Faster for frequently called methods.
- **Cons:** Slight initial delay due to compilation.

JIT Compilation Example in .NET

```
using System;

public class Program
{
   public static void Main()
   {
      Console.WriteLine("Hello, World!");
      int result = Add(5, 3);
      Console.WriteLine("Result: " + result);
   }

   public static int Add(int a, int b)
```

```
  {
    return a + b;
  }
}
```

Execution Process:

1. **Startup:** The CLR loads the assembly and inspects the metadata.
2. **Main Method:** The JIT compiler compiles the Main method into native code.
3. **Add Method:** When called, the Add method is compiled into native code.
4. **Reuse:** Subsequent calls to Add use the already compiled code.

Ahead-of-Time (AOT) Compilation

1. NGen (Native Image Generator)

- **Description:** Compiles IL code into native code before runtime and stores the compiled images in a cache.
- **Command:** ngen install MyApplication.exe

2. ReadyToRun (R2R)

- **Description:** Precompiles native code during the publish process, bundled with IL code for fallback.
- **Command:**

bash
Copy code
dotnet publish -c Release -r win-x64 -p:PublishReadyToRun=true

Comparison: AOT vs. JIT

Feature	AOT Compilation	JIT Compilation
Timing	Pre-runtime	Runtime
Performance	Faster startup	Dynamic optimizations
Portability	Platform-specific	Platform-independent
Use Cases	Startup-critical apps	General-purpose apps

is AOT and prejit same?

No, **Ahead-of-Time (AOT)** compilation and **Pre-JIT (Pre-Just-In-Time)** compilation are not the same, although they share some similarities. Here's a comparison to clarify:

1. Ahead-of-Time (AOT) Compilation

- **Definition:** AOT compiles Intermediate Language (IL) code into native machine code *before* the application is run.
- **When it Happens:** During the deployment or publish process (e.g., using tools like NGen or ReadyToRun in .NET).

- **Characteristics:**
  - Produces native machine code ahead of runtime, reducing runtime overhead.
  - The compiled native code is platform-specific and may not be portable.
  - Often includes IL alongside native code (e.g., ReadyToRun) for fallback to JIT when necessary.
- **Advantages:**
  - Faster application startup since runtime compilation is reduced or eliminated.
  - Predictable runtime performance.
- **Use Cases:** Applications where startup time is critical, or runtime environments where JIT is undesirable (e.g., environments with limited resources or constraints, like mobile or embedded systems).

2. Pre-JIT (Pre-Just-In-Time) Compilation

- **Definition:** Pre-JIT compiles all IL code into native code at the start of application execution, as part of the JIT compilation process.
- **When it Happens:** At runtime, during the startup phase of the application.
- **Characteristics:**
  - Part of the JIT family; happens in memory during the execution phase.
  - Native code is not stored permanently; it is generated dynamically and used during that execution session.
- **Advantages:**
  - Faster subsequent execution since all methods are precompiled at startup.
- **Disadvantages:**
  - Longer startup time because the entire application is compiled upfront.
- **Use Cases:** Scenarios where execution time is critical, but startup latency is acceptable.

Key Differences

Feature	AOT Compilation	Pre-JIT Compilation
Timing	Before runtime	During runtime startup
Persistence of Native Code	Native code is stored on disk or bundled with the application.	Native code is only retained in memory for the duration of execution.
Tool Examples	NGen, ReadyToRun	Handled by the CLR's JIT compiler
Portability	Limited (platform-specific native code)	High (IL code remains portable)
Startup Performance	Faster startup	Slower startup due to upfront compilation
Optimization	Limited runtime optimizations	Takes advantage of runtime data for optimizations

But same ngen is used for prejit and AOT?

Yes, **NGen (Native Image Generator)** can be used for both **Pre-JIT** and **AOT** compilation, but the way it is applied differs depending on the context. Here's how it works for each:

NGen and Pre-JIT

- **Pre-JIT Usage:** In this context, **NGen** can be used to compile the **entire application** into native code before execution, during the **installation** or **deployment** phase.
- **How it Works:**
  - When you run ngen install, it compiles the entire application's IL code into native code ahead of time.
  - The native images are stored in the **Native Image Cache** on the machine.
  - This process is **similar to Pre-JIT** in that it does all the compilation upfront, but NGen **stores** the generated native code so that it can be reused at runtime without the need for re-compilation.
  - This improves **startup time** because the application no longer needs to compile its code when executed.

NGen and AOT (Ahead-of-Time)

- **AOT Usage:** When used as an **AOT compiler**, **NGen** compiles the application into **native code before runtime**, and it is specific to a target machine.
- **How it Works:**
  - NGen generates **native images** from IL code ahead of time (before the application is run).
  - The compiled native images are stored in a special **Native Image Cache**, and the application uses these images during execution.
  - This eliminates the need for JIT compilation during runtime, which results in faster startup.
  - The generated native code is machine-specific (it works only on the machine it was compiled on).

Key Differences in NGen Usage

- **Pre-JIT Compilation with NGen:**
  - The native code is created during deployment.
  - It still relies on the CLR JIT compiler to compile code dynamically for methods that are called for the first time.
  - The focus is on **compiling all methods upfront** to reduce runtime overhead.
- **AOT with NGen:**
  - The entire application is compiled into native code, **before execution**.
  - The focus is on **eliminating JIT entirely** for the methods that have been precompiled.
  - It aims to improve **startup performance** and reduce runtime overhead by ensuring no JIT compilation is needed.

Conclusion

While **NGen** can be used in both **Pre-JIT** and **AOT** contexts, the purpose is slightly different:

- **Pre-JIT** focuses on compiling code upfront to optimize runtime performance.
- **AOT** focuses on completely eliminating JIT at runtime, providing faster startup and predictable performance by compiling code before execution.

In both cases, **NGen** serves the same purpose: **precompiling** IL code into **native machine code**.

Real time scenarios when these are used?

In real-time scenarios, **Pre-JIT** and **AOT** compilation strategies (including NGen) are used in different contexts to optimize application performance based on specific needs. Here are examples of real-world situations where **NGen**, **Pre-JIT**, and **AOT** might be used:

1. Pre-JIT in Real-Time Scenarios

Pre-JIT compilation is often used when **start-up performance** is important, and you want to reduce the overhead caused by JIT compilation during the application's initial execution. The NGen tool is typically used in these scenarios to generate native images for the application before it runs.

Real-Time Use Cases for Pre-JIT:

- **Enterprise Applications**:
  - **Scenario:** Large-scale enterprise applications with significant startup time.
  - **Example:** A company has a financial application that processes complex reports. The application is large, and JIT compilation during startup could make it slower. Using **Pre-JIT with NGen** to compile the entire application into native code during deployment significantly reduces the startup time, providing a smoother experience for users.
- **Client-Server Applications**:
  - **Scenario:** Applications with a client-server model that are frequently deployed on a consistent set of machines.
  - **Example:** A custom CRM or ERP solution that needs to be deployed on multiple machines. Pre-JIT with NGen ensures that all the necessary code is compiled during installation, minimizing runtime JIT overhead.
- **Web Applications (Server-side)**:
  - **Scenario:** Web applications where the server must quickly respond to user requests after starting up.
  - **Example:** A web API service hosted on an IIS server that serves thousands of requests. Pre-JIT with NGen compiles the IL code into native code during deployment, which helps speed up the server startup time.

2. AOT (Ahead-of-Time) Compilation in Real-Time Scenarios

AOT compilation is suitable for scenarios where **consistent performance**, **faster startup**, and **platform-specific optimization** are critical. AOT bypasses JIT compilation entirely and compiles the code into native machine code ahead of time.

Real-Time Use Cases for AOT:

- **Mobile Applications**:
  - **Scenario:** Mobile applications (especially in .NET MAUI or Xamarin) where startup performance and battery efficiency are crucial.
  - **Example:** A mobile app built using .NET MAUI might use **AOT** compilation to ensure that the application launches quickly and uses fewer resources during execution. AOT compilation ensures the app is precompiled into native machine code, reducing startup time and improving battery life by avoiding JIT compilation at runtime.
- **IoT (Internet of Things) Devices**:
  - **Scenario:** IoT devices have limited resources, and startup time and consistent performance are essential.

    - **Example:** A device running embedded .NET might use **AOT** compilation to ensure that the application is optimized for performance and doesn't require JIT compilation during runtime. This is important in embedded systems where runtime resources and memory are constrained, and quick startup is needed.
- **Cloud Native Applications**:
    - **Scenario:** Applications hosted in environments where fast scaling and predictable performance are necessary.
    - **Example:** A microservice built in .NET Core and deployed in a Kubernetes cluster might use **AOT** (e.g., using **ReadyToRun** or **NGen**) to compile the service ahead of time. This can reduce the cold start time when scaling the application, leading to quicker instance provisioning and more predictable performance across different environments.
- **Game Development**:
    - **Scenario:** Games or real-time applications where both startup speed and runtime performance are critical.
    - **Example:** A game developed in .NET that needs to load quickly to provide a seamless user experience can use **AOT** compilation. By precompiling the IL code into native machine code, the game can avoid the lag caused by JIT compilation, improving performance and reducing the initial loading time.
- **Enterprise Software on Desktops (Windows)**:
    - **Scenario:** Business applications that need to be deployed on a specific set of machines.
    - **Example:** A financial application that is used by employees in a company and needs to start up quickly and run with consistent performance throughout the day might use **AOT** compilation. Precompiling the application's IL code into native code ensures it runs efficiently, even on older hardware.

3. NGen in Real-Time Scenarios

While **Pre-JIT** and **AOT** are specific types of compilation strategies, **NGen** is the tool that is used for both. NGen can be used for **Pre-JIT** to precompile all IL code or for **AOT** to generate platform-specific native code ahead of time.

Real-Time Use Cases for NGen:

- **Application Deployment**:
    - **Scenario:** Deploying an application to a large number of machines, ensuring quick startup and consistent performance.
    - **Example:** When deploying a custom business application across multiple desktops in an enterprise, the use of **NGen** ensures the application starts quickly by having native code available from the start rather than JIT-compiling on the fly. This reduces deployment time and improves user experience during startup.
- **Gaming on Windows Platforms**:
    - **Scenario:** Games developed with Unity or .NET that require optimized startup times and efficient runtime performance.
    - **Example:** By using **NGen**, a game built using .NET Core for Windows can be precompiled for the target architecture. This results in faster load times and improved runtime performance.
- **ASP.NET Core Web Apps on IIS**:
    - **Scenario:** Web applications that need optimized startup times for a smooth user experience.

- **Example:** A web application built with **ASP.NET Core** that serves high volumes of traffic can benefit from **NGen** to precompile IL code into native code. This minimizes the runtime overhead of JIT compilation and provides faster application initialization.

Summary of Real-Time Use Cases:

- **Pre-JIT** (via NGen) is typically used when **startup performance** is a priority, such as in **large enterprise applications**, **client-server setups**, or **web APIs** that need quick responsiveness after startup.
- **AOT** (Ahead-of-Time compilation) is suitable for scenarios where **consistent performance** and **immediate startup** are critical, such as in **mobile apps**, **IoT devices**, **cloud-native microservices**, and **game development**.
- **NGen** can be employed in both Pre-JIT and AOT contexts to **precompile** code, ensuring faster startup times, more predictable performance, and less reliance on JIT compilation at runtime.

Managed Code vs. Unmanaged Code

Introduction

In the world of programming, **Managed Code** and **Unmanaged Code** are two categories of code that behave differently in terms of memory management, security, and execution. Understanding the distinction between these two types of code is important for developers, as each has its use cases, advantages, and limitations.

Managed Code

Managed code refers to the code that runs under the control of a **runtime environment**, such as the **Common Language Runtime (CLR)** in .NET. The CLR is responsible for many tasks, including garbage collection, memory management, type safety, and security enforcement.

Key Characteristics of Managed Code:

1. **Memory Management**: Managed code benefits from **automatic memory management** via garbage collection. The CLR takes care of memory allocation and deallocation, which reduces the risk of memory leaks.
2. **Security**: Managed code is subject to the security policies of the runtime environment (CLR). This includes type safety and ensuring that no invalid memory access occurs.
3. **Exception Handling**: Exception handling in managed code is uniform and integrated into the CLR. The system provides tools for handling and propagating exceptions.
4. **Interoperability**: Managed code can easily interact with other managed code and access high-level services (like database connections, file system operations) provided by the CLR.

Examples of Managed Code:

- **C#**: Code written in C# is managed code as it is compiled into Intermediate Language (IL) code, which is then executed by the CLR.

public class ManagedExample

```
{
   public static void Main()
   {
      int[] numbers = new int[10];
      numbers[0] = 100;
      Console.WriteLine("First Number: " + numbers[0]);
   }
}
```

- **VB.NET**: Similar to C#, code written in Visual Basic.NET is also considered managed code.

```
vb.net
Copy code
Module Module1
   Sub Main()
      Dim numbers(10) As Integer
      numbers(0) = 100
      Console.WriteLine("First Number: " & numbers(0))
   End Sub
End Module
```

- **F#**: Code written in F# is also managed code as it is part of the .NET ecosystem and runs within the CLR.

```
fsharp
Copy code
let numbers = [| 1; 2; 3; 4 |]
printfn "First number: %d" numbers.[0]
```

Advantages of Managed Code:

- **Automatic Memory Management**: With garbage collection, developers don't need to manually allocate and deallocate memory, reducing memory leaks and making memory management more efficient.
- **Security**: The CLR provides a robust security model, preventing common security issues like buffer overflow attacks.
- **Portability**: Managed code is platform-independent, as it can run on any machine that has a CLR, such as .NET Framework, .NET Core, or Xamarin.

Disadvantages of Managed Code:

- **Performance Overhead**: Garbage collection and runtime checks can introduce performance overhead, which might not be suitable for real-time or performance-critical applications.
- **Less Control**: Since the CLR manages memory and execution, developers have less direct control over low-level operations.

Unmanaged Code

Unmanaged code refers to code that runs directly on the hardware or the operating system without the control of a runtime environment like the CLR. Unmanaged code is typically written in languages such as **C**, **C++**, or **Assembly**.

Key Characteristics of Unmanaged Code:

1. **Memory Management**: In unmanaged code, the programmer is responsible for explicitly managing memory. This includes allocating memory and deallocating it when no longer needed. Failure to do so can lead to memory leaks or access violations.

2. **No Runtime Support**: Unmanaged code operates directly with system resources and does not rely on a runtime environment for garbage collection, type safety, or security.
3. **Performance**: Unmanaged code often provides better performance as it allows fine-grained control over system resources and doesn't incur overhead from a runtime environment.
4. **System-Level Operations**: Unmanaged code can interact with hardware, system APIs, and perform low-level operations, which are typically not available to managed code.

Examples of Unmanaged Code:

- **C**: C is one of the most common unmanaged languages. In C, memory must be managed manually using functions like malloc and free.

c
Copy code
```
#include <stdio.h>

int main() {
  int *arr = (int*)malloc(10 * sizeof(int));
  arr[0] = 100;
  printf("First Number: %d\n", arr[0]);
  free(arr);
  return 0;
}
```

- **C++**: C++ offers direct memory manipulation and has both high-level and low-level features.

cpp
Copy code
```
#include <iostream>
using namespace std;

int main() {
  int* arr = new int[10];
  arr[0] = 100;
  cout << "First Number: " << arr[0] << endl;
  delete[] arr;
  return 0;
}
```

- **Assembly**: Assembly language is the lowest-level programming language, providing complete control over the hardware.

Advantages of Unmanaged Code:

- **Performance**: Unmanaged code often runs faster than managed code because it does not have the overhead of garbage collection or runtime checks.
- **Control**: Developers have complete control over memory allocation, hardware access, and system resources, making it ideal for system programming, device drivers, or real-time applications.
- **Low-Level Operations**: Unmanaged code allows direct interaction with hardware and system-level APIs, making it necessary for developing operating systems, embedded systems, or performance-critical applications.

Disadvantages of Unmanaged Code:

- **Memory Leaks**: Developers must manually manage memory. Incorrect memory management can lead to memory leaks, buffer overflows, and segmentation faults.
- **Security Risks**: The lack of runtime checks makes unmanaged code more susceptible to security vulnerabilities like buffer overflow attacks.
- **Platform Dependence**: Unmanaged code is often platform-specific, as it requires direct interaction with the operating system or hardware.

Managed Code vs. Unmanaged Code: Comparison

Feature	Managed Code	Unmanaged Code
Execution	Runs under the control of a runtime (e.g., CLR).	Runs directly on the system's hardware or OS.
Memory Management	Automatic memory management via garbage collection.	Manual memory management (malloc, free).
Security	Type safety, automatic bounds checking, and runtime checks.	No automatic security checks, more prone to security vulnerabilities.
Performance	Slower due to the overhead of the runtime environment.	Faster, with more control over resources.
Control	Limited control over system resources.	Full control over system resources.
Languages Used	C#, VB.NET, F#, etc.	C, C++, Assembly, etc.

Conclusion

Both **Managed Code** and **Unmanaged Code** have their places in modern software development. **Managed Code** is ideal for high-level applications where security, ease of development, and memory management are priorities. **Unmanaged Code**, on the other hand, is better suited for low-level systems programming, performance-critical applications, and situations where developers need direct access to hardware or system resources.

Understanding when and where to use these types of code can significantly impact the performance, security, and maintainability of your software.

Exe and DLL: Assemblies and Namespaces in .NET

Introduction

In .NET, the concepts of **EXE** and **DLL**, as well as **Assemblies** and **Namespaces**, are central to the organization and deployment of applications. Understanding how they work together helps developers structure, organize, and manage their code effectively. Let's break down these concepts in detail.

EXE and DLL

EXE (Executable File)

- **Definition**: An **EXE** (executable) is a type of file that contains compiled code that can be executed directly by the operating system. When you run an application in Windows (e.g., a desktop app or a console application), you are running an EXE file.
- **Characteristics**:
  - An EXE file is typically the entry point of a .NET application (such as a Console Application or Windows Forms Application).
  - It can be run directly by the operating system.
  - **Example**: MyApp.exe
- **Usage**:
  - Used to package applications that the user interacts with directly.
  - An EXE file can contain **one assembly** with executable code.
- **Example in .NET**:

```
// Program.cs in a Console Application
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Hello, World!");
   }
}
```

This will produce Program.exe, which can be executed to run the application.

DLL (Dynamic Link Library)

- **Definition**: A **DLL** is a file that contains compiled code, but unlike EXE files, it cannot be executed directly. DLLs are used to store reusable code, libraries, or components that can be shared across different applications or parts of an application.
- **Characteristics**:
  - A DLL does not have an entry point (like Main() in an EXE).
  - It is referenced by EXE files or other DLLs to provide functionality.
  - **Example**: MyLibrary.dll
- **Usage**:
  - DLLs contain classes, functions, and resources that can be used by other applications or modules.
  - Commonly used for **reusable code**, **shared libraries**, or **frameworks**.
- **Example in .NET**:

```
// Library.cs in a class library project
public class Calculator
{
   public int Add(int a, int b) => a + b;
}
```

This will produce Library.dll, which can be referenced by EXE files or other DLLs.

Assemblies in .NET

What is an Assembly?

- **Definition**: An **Assembly** is a compiled unit of code in the .NET framework that consists of one or more files. These files can be **EXE** or **DLL** files, and they contain metadata, code, and resources needed for the application to run.
- **Key Points**:
    - Assemblies are the **building blocks** of a .NET application.
    - They include metadata about the types and resources contained within the assembly, such as versioning, security, and other attributes.
    - An assembly can contain multiple **classes** and **namespaces**.
    - Every .NET application consists of at least one assembly, typically an EXE file for the entry point or a DLL for shared components.

Types of Assemblies:

1. **Private Assemblies**: Used by a single application and stored in the application's directory.
2. **Shared Assemblies**: Can be used by multiple applications and stored in the Global Assembly Cache (GAC).
3. **Dynamic Assemblies**: Created at runtime and typically used for scenarios where code is generated dynamically (e.g., Reflection.Emit).

Example:

Consider an assembly MyApp.dll that contains multiple classes and namespaces:

```
// Inside MyApp.dll

namespace MyApp.Utilities
{
  public class Calculator
  {
    public int Add(int a, int b) => a + b;
  }
}

namespace MyApp.Services
{
  public class EmailService
  {
    public void SendEmail(string email)
    {
      // Logic to send email
    }
  }
}
```

Here, MyApp.dll contains two namespaces: MyApp.Utilities and MyApp.Services, and the classes Calculator and EmailService are logically grouped under these namespaces.

Namespaces in .NET

What is a Namespace?

- **Definition**: A **Namespace** is a logical container for classes, interfaces, structs, and other types. It is used to **organize code** and avoid naming conflicts by grouping related types together.
- **Key Points**:

- Namespaces provide **logical grouping** of classes and can contain **sub-namespaces**.
- A single assembly can contain multiple namespaces.
- Namespaces help in avoiding naming conflicts, as different parts of an application may use the same class names, but with different namespaces.

- **Structure**:
  - A namespace can have any number of classes or sub-namespaces.
  - The classes within a namespace can represent different functionalities, like database access, utilities, and services.

Example:

In the previous example, we have two namespaces: MyApp.Utilities and MyApp.Services. Each namespace contains different classes.

```
namespace MyApp.Utilities
{
  public class Logger
  {
    public void Log(string message)
    {
      Console.WriteLine(message);
    }
  }
}

namespace MyApp.Services
{
  public class PaymentService
  {
    public void ProcessPayment(string paymentDetails)
    {
      Console.WriteLine("Processing payment...");
    }
  }
}
```

- Here, Logger is logically grouped under the MyApp.Utilities namespace.
- PaymentService is logically grouped under the MyApp.Services namespace.

Benefits of Namespaces:

1. **Avoiding Name Conflicts**: Multiple classes with the same name can exist in different namespaces without causing conflicts.
2. **Organization**: Namespaces help to logically organize code, making it easier to understand and maintain.
3. **Hierarchical Grouping**: Namespaces can be nested to create hierarchical structures, such as System.IO for input/output operations.

Relationship Between Assemblies and Namespaces

- **One Assembly Can Contain Multiple Namespaces**: An assembly can consist of one or more namespaces, and each namespace contains classes, interfaces, or other types.

Example: In the previous example, MyApp.dll contained MyApp.Utilities and MyApp.Services as namespaces, which further contained various classes like Calculator, Logger, and PaymentService.

- **Namespaces Group Code Logically**: While assemblies are physical files (EXE or DLL), namespaces are a logical construct that help organize the code within those assemblies.

Conclusion

To summarize, in .NET:

- **EXE** files are executable assemblies that run applications.
- **DLL** files are reusable libraries that contain code to be shared among multiple applications.
- **Assemblies** are compiled units that can contain one or more files (EXE or DLL) and contain metadata and code for the application.
- **Namespaces** are logical groupings of classes and types, helping organize code and avoid naming conflicts.

```
using System;

public class Class1
{
static int Main(string[] args)
{
Console.WriteLine("Hello C#");
return 0;
}
}
```

myorg.cs

```
namespace MyOrg
{
public class Class1
{
public string Hello()
{
return "Hello:Class1";
}
}
namespace Finance
{
public class Class2
{
public string Greet()
```

```
{
return "Greet:Class2";
}
}
namespace Payroll
{
public class Class3
{
public string Welcome()
{
return "Welcome:Class3";
}
}
}
}
}

csc -t:library myorg.cs
```

```
using MyOrg;
using MyOrg.Finance;
using System;
using Payroll= MyOrg.Finance.Payroll;
public class MainClass
{
static void Main()
{
Class1 c1 = new Class1();
string s1=c1.Hello();
Console.WriteLine(s1);


Console.WriteLine(new Class2().Greet());
Console.WriteLine(new Payroll.Class3().Welcome());

}
}
csc -r:Epam.dll Demo3.cs
```

.NET Does not support primitive Data types

In .NET, **primitive data types** as commonly referred to in other programming languages like C or C++ do not exist in the same way. However, .NET provides **value types** and **reference types**, which are analogous to primitive data types. Let's clarify this concept:

Primitive Data Types in .NET

.NET does support **built-in types** that are commonly considered as primitive data types in other languages. These types are **value types** and are part of the .NET Framework's **System** namespace.

Common Value Types (Analogous to Primitive Data Types)

1. **Integer Types**:
   - int (alias for System.Int32) — A 32-bit signed integer.
   - long (alias for System.Int64) — A 64-bit signed integer.
   - short (alias for System.Int16) — A 16-bit signed integer.
   - byte (alias for System.Byte) — An 8-bit unsigned integer.
2. **Floating Point Types**:
   - float (alias for System.Single) — A 32-bit single-precision floating-point number.
   - double (alias for System.Double) — A 64-bit double-precision floating-point number.
3. **Other Numerical Types**:
   - decimal (alias for System.Decimal) — A 128-bit precise decimal type used for financial and monetary calculations.
4. **Boolean Type**:
   - bool (alias for System.Boolean) — Represents a true or false value.
5. **Character Type**:
   - char (alias for System.Char) — Represents a 16-bit Unicode character.
6. **Date and Time Types**:
   - DateTime (alias for System.DateTime) — Represents a date and time value.
7. **Struct Types**:
   - struct (e.g., System.Guid) — User-defined types that can contain multiple values but are still value types.

These types are **value types**, meaning they directly hold data. The default values for these types (e.g., 0 for numbers, false for bool) are allocated on the stack.

What .NET Does Differently

- **No Direct Equivalents for Primitive Types**: While languages like C/C++ have "primitive" types like int, float, etc., .NET uses the concept of **value types** and **reference types**. However, .NET still provides similar types under the System namespace (e.g., System.Int32, System.Double, etc.).
- **Boxing and Unboxing**: In .NET, **value types** (such as int, bool, etc.) are typically stored on the **stack**, and when assigned to a **reference type** (like an object), the value type is boxed into the heap. This process is called **boxing**. To convert it back to its original value type, **unboxing** is required.

```
int x = 42;       // Value type stored on the stack
object obj = x;   // Boxing: stored on the heap
int y = (int)obj; // Unboxing: converting back to a value type
```

- **Reference Types**: On the other hand, **reference types** (like string, arrays, classes, etc.) are stored in the **heap** and hold a reference (or pointer) to the data, unlike **value types**.

Conclusion

To sum up:

- While **.NET** does not have "primitive data types" in the same sense as C/C++, it provides **value types** (e.g., int, float, char, bool) that are functionally similar to primitive types in other languages.
- **Assemblies** in .NET contain these built-in types (which are part of the system) and user-defined types (such as classes and structs).

Thus, **.NET** supports data types that are "primitive" in functionality but categorizes them under value types and reference types, providing flexibility and enhanced type safety.

```
using System;

public class Class1
{
static void Main()
{
Console.WriteLine("Enter the first number");
string s1=Console.ReadLine();
Console.WriteLine("Enter the second number");
string s2=Console.ReadLine();
int i1=Int32.Parse(s1);
int i2=int.Parse(s2);
Console.WriteLine((i1+i2));
}
}
```

C# Language basics

1. C# Is Object-Oriented

C# is an **object-oriented programming** (OOP) language, which means it supports the core principles of OOP such as:

- **Encapsulation**: Grouping related data and methods that operate on the data into a single unit known as a class.
- **Inheritance**: Allowing a new class to inherit the properties and behaviors of an existing class.
- **Polymorphism**: Allowing different classes to provide different implementations of the same method or interface.
- **Abstraction**: Hiding the complex implementation details from the user and providing a simple interface.

2. Case Sensitivity and Naming Conventions

C# is **case-sensitive**, meaning that variables and methods with the same name but different cases are considered different. For example, myVariable and MyVariable are distinct variables.
Naming conventions in C#:

- **PascalCasing**: This convention is used for public identifiers (e.g., class names, method names, properties).
    - Example: MyClass, GetDetails(), CustomerName
- **camelCasing**: This convention is used for local variables, private fields, and method parameters.
    - Example: myVariable, _userId, firstName

3. Comments
C# supports different types of comments:

- **Single-line comments**: Begin with // and extend to the end of the line.

```
// This is a single-line comment
```

Multi-line comments: Begin with /* and end with */.

```
/* This is a
  multi-line comment */
```

XML comments: Used to generate documentation for classes, methods, etc. They begin with ///.

```
/// <summary>
/// This method calculates the total price.
/// </summary>
public decimal CalculateTotal() { ... }
```

4. C# is a Strongly Typed Language
C# is a **strongly typed** language, which means that every variable and constant must have a specified type at compile time. You cannot assign a value of one type to a variable of another type unless explicit conversion is done.
For example, trying to assign a string to an integer will result in a compile-time error:

```
int myNumber = "Hello"; // Error: Cannot implicitly convert type 'string' to 'int'
```

5. Variable Declaration
In C#, variables are declared with a specific type, followed by the variable name and an optional initial value. The general syntax is:
<datatype> <varname> = <default value>;

For example:

```
int age = 30;  // Declaring an integer variable with an initial value of 30
string name = "John";  // Declaring a string variable with an initial value of "John"
bool isActive = true;  // Declaring a boolean variable with an initial value of true
```

You can also use var to let the compiler infer the type based on the assigned value:

```
var score = 99; // Implicitly inferred as int
var userName = "Alice"; // Implicitly inferred as string
```

6. Method Declaration
A **method** in C# is a block of code that performs a specific task. The general syntax for declaring a method is:

```
<returntype> <name>(<paramList>)
{
  // Method body
}

public int AddNumbers(int num1, int num2)
{
  return num1 + num2;
}
```

Example: Putting It All Together

```
using System;

public class Program
{
  // Main method: entry point of the application
  public static void Main(string[] args)
  {
    // Variable declaration with PascalCasing for method name
    string userName = "John";
    int userAge = 30;

    // Method call with camelCasing for local variables
    string result = GetUserInfo(userName, userAge);

    // Display the result
    Console.WriteLine(result);
  }

  // Method declaration with PascalCasing for method name
  public static string GetUserInfo(string name, int age)
  {
    return $"User's name is {name} and age is {age}.";
  }
}
```

C# Supports 3 Parameter Passing Mechanisms:

1. **Pass by Value (Default):**
   - ✔ **This is correct. By default, C# uses pass-by-value for parameters.**
     - **When a parameter is passed by value, the parameter and argument are stored in different memory locations.**
     - **Manipulations to the parameter inside the method do not affect the original argument outside the method.**

     Example:

```
void Increment(int value)
{
  value++;
}

int number = 10;
Increment(number); // number remains 10 because only a copy is modified.
```

2. **Pass by Reference (ref):**
   - ✔ **This is mostly correct but needs more explanation.**
     - **When a parameter is passed by reference using the ref keyword, the parameter and argument refer to the same memory location.**
     - **Changes made to the parameter inside the method will directly affect the original argument outside the method.**
     - **The ref keyword requires that the variable being passed is initialized before the method call.**

Example:

```
void Increment(ref int value)
{
  value++;
}

int number = 10;
Increment(ref number); // number becomes 11 because the same memory is modified.
```

3. **Pass by Out (out):**
   - ✔ **The explanation is partially correct but requires clarification.**
     - **The out keyword is similar to ref, but it is specifically used for returning output values from a method.**
     - **Unlike ref, the variable passed with out does not need to be initialized before the method call, but it must be assigned a value inside the method before returning.**
     - **The key distinction is that out parameters are intended solely for output, whereas ref parameters allow both input and output.**

Example:

```
void GetResult(out int result)
{
  result = 100; // Must assign a value to `result` before exiting the method.
}

int value;
GetResult(out value); // value becomes 100.
```

Explanation:

C# supports 3 parameter-passing mechanisms:

1. **Pass by Value (Default): The parameter and argument have different memory locations, so changes to the parameter inside the method do not affect the original argument.**

2. **Pass by Reference (ref): The parameter and argument share the same memory location, so changes to the parameter inside the method affect the original argument.**
3. **Pass by Out (out): Similar to ref, but the parameter is used only to return output values. The variable does not need to be initialized before being passed but must be assigned inside the method.**

```
using System;

public class Class1
{
public void Display(out int i)
{
i=20;
i=i+1;
Console.WriteLine(i);
}
static void Main()
{
int j=10;
new Class1().Display(out j);
Console.WriteLine(j);
}
}
```

what about in parameter. will it come under this classification?

The in parameter in C# does not fall under the traditional classification of **Pass by Value**, **Pass by Reference**, or **Pass by Out**, as it has a distinct purpose and behavior. However, it is related to **Pass by Reference** in how it operates. Here's an explanation:

in Parameter in C#

The in modifier in C# is used to pass arguments **by reference**, but with a restriction that the parameter is **read-only** inside the method. This means:

- The in parameter allows you to pass large objects (like structs) by reference to avoid the performance cost of copying, **while ensuring the method cannot modify the argument**.
- Unlike ref, the argument passed with in **cannot be modified** inside the method.
- The variable being passed with in **must be initialized** before the method call, just like with ref.

Key Characteristics of in:

1. **Similarities to ref:**
    - Both pass the argument **by reference**, so they share the same memory location.
    - Both require the variable to be initialized before being passed.

2. **Differences from ref:**
    - in parameters are **read-only** inside the method, while ref allows modification.
    - Methods cannot assign a new value to an in parameter.
3. **Differences from out:**
    - out parameters are **only for output**, while in is for **read-only input**.
    - out does not require initialization before the method call, but in does.

Example of in Parameter:

```
void PrintLength(in string value)
{
   Console.WriteLine(value.Length); // Allowed
   // value = "new value"; // Compilation error, `in` makes `value` read-only.
}

string name = "John";
PrintLength(in name); // Passes `name` by reference for read-only access.
```

Should in Be Included in the Classification?
While in is technically a **reference-passing mechanism**, it is distinct because it imposes a **read-only restriction**. For completeness, you can add it as a fourth parameter-passing mechanism in C#, as it provides unique functionality that complements the existing ones:

1. **Pass by Value (Default):** Copies the value.
2. **Pass by Reference (ref):** Allows modifications.
3. **Pass by Out (out):** Only for output.
4. **Pass by Reference (Read-Only, in):** Passes by reference but does not allow modifications.

Optional Parameters in C#

In C#, **optional parameters** allow you to define parameters with default values in method declarations. This way, when calling the method, you can choose to omit the argument for those optional parameters, and the default value will be used instead.
Syntax for Optional Parameters
Optional parameters are specified by providing a default value for the parameter in the method signature:
<returntype> <methodname>(<parameter1>, <parameter2> = <default value>, <parameter3> = <default value>)

parameter2 = <default value> specifies that parameter2 is optional, and if not passed when calling the method, it will use the default value.

Example 1: Using Optional Parameters
Here's an example demonstrating how optional parameters work:

```
public class Calculator
```

```
{
   // Method with optional parameters b and c
   public int Add(int a, int b = 0, int c = 0)
   {
      return a + b + c;
   }
}

public class Program
{
   public static void Main(string[] args)
   {
      Calculator calculator = new Calculator();

      // Call with all arguments
      Console.WriteLine(calculator.Add(1, 2, 3));  // Output: 6

      // Call with two arguments
      Console.WriteLine(calculator.Add(1, 2));    // Output: 3 (c uses default value 0)

      // Call with one argument
      Console.WriteLine(calculator.Add(1));       // Output: 1 (b and c use default values 0)
   }
}
```

Important Rule: Optional Parameters Must Appear After Required Parameters

Optional parameters must always appear after required parameters in the method declaration. If you try to declare optional parameters before required ones, the compiler will throw an error.

Incorrect Example:

```
public int Mul(int a = 0, int b, int c = 0)  // Error: Optional parameters cannot appear before required parameters
{
   return a * (b * c);
}
```

Corrected Example:

```
public int Mul(int b, int a = 0, int c = 0)
{
   return a * (b * c);
}
```

Named Arguments

Console.WriteLine(calculator.Add(c:1, b:2, a:3));

1. When a method is written twice with the same parameter list , but differs in return type - is it overloading?
    1. Ex: public int Add(int i, int j)
       {
          return i + j;
       }
       public double Add(int i, int j)
       {
          return (double) i + j;
       }

    No, by default it returns the error, saying it as a duplicate, it should differ in parameter signature in some way ex: double Add(int i, in int j) - Just like Out and Ref keyword, in keyword accepts only input, either you can use in or ref in this case if needed
        2. In keyword is – pass by reference and read only
        3. Out keyword – pass by reference and write only

Conditional Structures in C#

Conditional structures are used to control the flow of execution based on conditions. The most commonly used conditional statements in C# are if, else if, else, and switch.

1. if / else if / else
The if statement is used to test a condition. If the condition is true, the block of code inside the if statement is executed. If it's false, and there are additional else if or else statements, they are evaluated.

```
if (condition1)
{
  // Code block executed if condition1 is true
}
else if (condition2)
{
  // Code block executed if condition1 is false and condition2 is true
}
else
{
  // Code block executed if none of the conditions are true
}
```

Example:

```
int number = 10;
```

```
if (number > 0)
{
  Console.WriteLine("The number is positive.");
}
else if (number < 0)
{
  Console.WriteLine("The number is negative.");
}
else
{
  Console.WriteLine("The number is zero.");
}
```

2. switch Statement

The switch statement is used to test a variable or expression against multiple possible values. It can be a more efficient way to handle multiple conditions based on a single expression.

Syntax:

```
switch (expression)
{
  case value1:
    // Code block executed if expression == value1
    break;
  case value2:
    // Code block executed if expression == value2
    break;
  case value3:
    // Code block executed if expression == value3
    break;
  default:
    // Code block executed if no case matches
}
```

Example:

```
int day = 3;

switch (day)
{
  case 1:
    Console.WriteLine("Monday");
    break;
  case 2:
    Console.WriteLine("Tuesday");
    break;
  case 3:
    Console.WriteLine("Wednesday");
```

```
    break;
  case 4:
    Console.WriteLine("Thursday");
    break;
  case 5:
    Console.WriteLine("Friday");
    break;
  default:
    Console.WriteLine("Weekend");
    break;
}
```

Loop Structures in C#

Looping structures are used to repeat a block of code as long as a condition is true. The most common loop types in C# are while, do-while, for, and foreach.

1. while Loop

The while loop repeatedly executes a block of code as long as the specified condition is true.

```
while (condition)
{
  // Code block executed as long as condition is true
}
```

Example:

```
int i = 1;
while (i <= 5)
{
  Console.WriteLine(i);
  i++;
}
```

2. do-while Loop

The do-while loop is similar to the while loop, but it guarantees that the code inside the loop will run at least once, even if the condition is false.

```
do
{
  // Code block executed at least once, and then repeats as long as condition is true
} while (condition);
```

Example:

```
int i = 1;
do
{
   Console.WriteLine(i);
   i++;
} while (i <= 5);
```

3. for Loop

The for loop provides a concise way to initialize, test, and modify the loop variable in a single statement.

```
for (initialization; condition; increment/decrement)
{
   // Code block executed as long as condition is true
}
```

```
for (int i = 1; i <= 5; i++)
{
   Console.WriteLine(i);
}
```

4. foreach Loop

The foreach loop is used to iterate over elements in a collection (e.g., arrays, lists, or other collections). It simplifies the syntax for looping through each element without needing an index or counter.

```
foreach (var element in collection)
{
   // Code block executed for each element in the collection
}
```

```
int[] numbers = { 1, 2, 3, 4, 5 };
foreach (var number in numbers)
{
   Console.WriteLine(number);
}
```

Using Aliases in C#

using aliasName = fullyQualifiedName;

Example: Using Aliases with Namespaces

```
using a = System.Console;
```

```
class Program
{
   static void Main()
   {
      // Using the alias 'a' instead of the full 'System.Console'
      a.WriteLine("Hello, World!");
   }
}
```

Using aliases in C# is a helpful tool to:

- **Shorten long namespaces or type names.**
- **Resolve conflicts between types with the same name.**
- **Improve code readability and maintainability.**

System namespace

Object (Supermost class)
Console
- ValueType
- - SByte
- - Int16
Int32
Int64
Byte
UInt16
UInt32
UInt64
Single
Double
Array
Exception
String
Char
Boolean
Decimal
Datetime
....

Arrays in C#

An **array** in C# is a collection of elements, all of which must be of the same type. Arrays are indexed starting from 0, meaning the first element is at index 0, the second at index 1, and so on.

Arrays in C# are objects, and they are instances of the System.Array class, which provides various methods for working with arrays.

Declaring Arrays

There are several ways to declare and initialize arrays in C#. Let's look at the different ways to declare arrays:

1. **Using new Keyword**

You can declare an array and initialize it using the new keyword, specifying the size of the array:

```
int[] i = new int[2]; // Declare an array of 2 integers
i[0] = 23;          // Assign value to first element
i[1] = 45;          // Assign value to second element
```

2. **Array Initialization Using Curly Braces**

If you want to initialize an array with specific values at the time of declaration, you can use curly braces {}:

```
int[] i = new int[] { 23, 45 }; // Declare and initialize with values
```

Implicit Array Initialization

You can also omit the new int[] part and just use curly braces to initialize the array:

```
int[] i = { 23, 45 }; // Implicit initialization
```

Array as an Object

When you declare an array, it is internally treated as an object of the System.Array class, which provides several useful methods, such as Length, GetLength(), and Clone(). Here's an example of accessing the Length property:

```
int[] i = { 23, 45 };
Console.WriteLine("Array length: " + i.Length); // Output: 2
```

Accessing Array Elements

```
int[] i = { 23, 45 };
Console.WriteLine(i[0]); // Output: 23
Console.WriteLine(i[1]); // Output: 45
```

Array Methods and Properties

As System.Array class objects, arrays provide a number of useful methods and properties to interact with arrays:

- **Length: Returns the total number of elements in the array.**
- **GetLength(dimension): Returns the size of the array in a specific dimension (useful for multi-dimensional arrays).**
- **Clone(): Creates a shallow copy of the array.**

```
int[] i = { 23, 45, 67 };
Console.WriteLine(i.Length);  // Output: 3
int[] copy = (int[])i.Clone();  // Create a shallow copy
Console.WriteLine(copy[0]); // Output: 23
```

Multi-Dimensional Arrays

In C#, you can also declare multi-dimensional arrays (e.g., 2D arrays):

```
int[,] matrix = new int[2, 2]; // Declare a 2D array (2 rows, 2 columns)
matrix[0, 0] = 1;
matrix[0, 1] = 2;
matrix[1, 0] = 3;
matrix[1, 1] = 4;

Console.WriteLine(matrix[1, 1]); // Output: 4
```

Jagged Arrays

A jagged array is an array of arrays. Each element of a jagged array can hold another array, allowing for non-rectangular multi-dimensional structures.

```
int[][] jaggedArray = new int[2][];
jaggedArray[0] = new int[] { 1, 2 };
jaggedArray[1] = new int[] { 3, 4, 5 };

Console.WriteLine(jaggedArray[1][2]); // Output: 5
```

Assignment

1. Find the Largest Element in an Array

Problem Statement:

You are given an array of integers. Your task is to find the largest element in the array.
Write a function findLargest that returns the largest element.

Method Template:

```
public class Solution {
   public int FindLargest(int[] arr) {
      // Your code here
   }
}
```

2. Find the Second Largest Element in an Array

Problem Statement:

You are given an array of integers. Your task is to find the second largest element in the array.
Write a function findSecondLargest that returns the second largest element.

Method Template:

```
public class Solution {
   public int FindSecondLargest(int[] arr) {
      // Your code here
```

```
    }
}
```

3. Check if an Array is Sorted

Problem Statement:

Given an array of integers, determine if the array is sorted in ascending order. If it is sorted, return true; otherwise, return false.

Write a function isArraySorted that returns a boolean indicating whether the array is sorted in ascending order.

Method Template:

```
public class Solution {
    public bool IsArraySorted(int[] arr) {
        // Your code here
    }
}
```

4. Reverse an Array

Problem Statement:

Given an array of integers, reverse the array in place (do not use any extra space).

Write a function reverseArray that reverses the elements of the array.

Method Template:

```
public class Solution {
    public void ReverseArray(int[] arr) {
        // Your code here
    }
}
```

5. Move Zeros to the End of an Array

Problem Statement:

You are given an array of integers. Your task is to move all zeros to the end of the array without changing the order of non-zero elements.

Write a function moveZerosToEnd that moves all zeros to the end of the array.

Method Template:

```
public class Solution {
    public void MoveZerosToEnd(int[] arr) {
        // Your code here
    }
}
```

6. Find the Missing Number in an Array

Problem Statement:

You are given an array containing n-1 integers from the range 1 to n. One number from this range is missing. Find and return the missing number.

Write a function findMissingNumber that returns the missing number.

Method Template:

```
public class Solution {
   public int FindMissingNumber(int[] arr, int n) {
      // Your code here
   }
}
```

7. Find Common Elements in Two Sorted Arrays

Problem Statement:

You are given two sorted arrays of integers. Your task is to find the common elements between both arrays.

Write a function findCommonElements that returns a list of common elements between both arrays.

Method Template:

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

public class Solution {
   public List<int> FindCommonElements(int[] arr1, int[] arr2) {
      // Your code here
   }
}
```

Param Arrays

When we declare an array parameter as the paramarrray, that parameter can accept any no of arguments of that parameter type.

```
using System;

public class Class1
{
public int Add(params int[] ia)
{
int sum=0;
for(int i=0; i<ia.Length;i++)
{
sum+=ia[i];
}
return sum;
}
static void Main(string[] args)
{
Class1 c1= new Class1();
Console.WriteLine(c1.Add(12, 23, 45));
Console.WriteLine(c1.Add(13,24,56,56,6,6,76,76));
}
}
```

String formatting

```
using System;

public class Class1
{
static void Main(string[] args)
{
try
{
string s1=args[0];
int i1=Int32.Parse(s1);
string s2=args[1];
int i2=int.Parse(s2);
Console.WriteLine("The sum of " + i1 + " and " + i2 + " is:"  + (i1+i2));
Console.WriteLine("The sum of {0} and {1} is: {2}", i1, i2, (i1+i2));

}
catch(IndexOutOfRangeException ex)
{
Console.WriteLine(ex.Message);
}
catch(FormatException ex)
{
Console.WriteLine(ex.Message);
}
catch(Exception ex)
{
Console.WriteLine(ex.Message);
}
finally
{
Console.WriteLine(”in finally”);

}
Console.WriteLine("response from API");
}
}
```

String verbatim

In C#, a verbatim string literal is a type of string literal that preserves **whitespace** and allows the **inclusion of escape characters** without the need for escaping them.

Verbatim string literals are prefixed with the **@** symbol.

```
using System;

class Program
{
   static void Main()
   {
      string multiLineString = @"
         This is a multi-line string.
         It preserves all white space
         and line breaks exactly as typed.";

      Console.WriteLine(multiLineString);
   }
}
```

String Interpolation

String interpolation in C# provides a more readable and convenient way to format strings compared to traditional methods like **String.Format**

Interpolated strings are prefixed with a **$** and allow embedding expressions inside curly braces **{}** directly within the string.

```
using System;

class Program
{
   static void Main()
   {
      string name = "John";
      int age = 30;

      string greeting = $"Hello, my name is {name} and I am {age} years old.";
      Console.WriteLine(greeting); // Output: Hello, my name is John and I am 30 years old.
   }
}
```

typeof Operator

```
using System;

public class Class1
{
            static void Main()
            {
```

```
            Type t = typeof(Class1);
            Console.WriteLine(t.Name);
            Console.WriteLine(t.Assembly.FullName);
        }
}
```

ternary operator

The **ternary operator** is a concise way of performing conditional evaluations. It acts as a shorthand for simple if-else statements and is particularly useful for returning values based on a condition.
The syntax of the ternary operator is as follows:
csharp
Copy code
condition ? valueIfTrue : valueIfFalse;

- **condition**: The expression that is evaluated (a boolean expression).
- **valueIfTrue**: The value returned if the condition is true.
- **valueIfFalse**: The value returned if the condition is false.

Example 1: Basic Ternary Operator Usage

```
int a = 5, b = 10;
int larger = (a > b) ? a : b;
Console.WriteLine(larger); // Output: 10
```

Example 2: Using Ternary Operator for Null Check

```
string name = null;
string message = (name != null) ? "Name is provided" : "Name is null";
Console.WriteLine(message); // Output: Name is null
```

Example 3: Multiple Ternary Operators

```
int x = 15;
string result = (x > 10) ? "Greater than 10" : (x == 10) ? "Equal to 10" : "Less than 10";
Console.WriteLine(result); // Output: Greater than 10
```

Nullable Types in C#

In C#, **nullable types** allow value types (such as int, bool, DateTime, etc.) to be assigned a value of null. This is useful when dealing with databases or other scenarios where a value might be missing or undefined.
Declaring Nullable Types
You can make a value type nullable by appending a ? to the type declaration. This allows the variable to hold a null value as well as its usual values.

```
int? nullableInt = null;
bool? nullableBool = true;
DateTime? nullableDateTime = null;
```

Working with Nullable Types

1. **HasValue Property**: This property checks whether the nullable type contains a value. It returns true if the nullable type has a value, and false otherwise.
2. **Value Property**: This property retrieves the value of the nullable type if it exists. However, if the nullable type is null, trying to access the Value property will throw an exception (InvalidOperationException).
3. **Default Values**: You can use the null-coalescing operator (??) to assign a default value if the nullable type is null.

null-coalescing operator

The null-coalescing operator provides a default value if the left-hand operand is **null**.

```
int? nullableInt = null;
int defaultInt = nullableInt ?? 5;
```

```
using System;

class Program
{
   static void Main()
   {
      // Declare nullable variables
      int? nullableInt = null;
      bool? nullableBool = true;
      DateTime? nullableDateTime = null;

      // Check if nullableInt has a value
      if (nullableInt.HasValue)
         Console.WriteLine($"nullableInt has a value: {nullableInt.Value}");
      else
         Console.WriteLine("nullableInt is null.");

      // Using null-coalescing operator to provide a default value
      int defaultInt = nullableInt ?? 0; // If nullableInt is null, assign 0
      Console.WriteLine($"defaultInt: {defaultInt}"); // Output: defaultInt: 0

      // Example of using nullable bool
      Console.WriteLine($"nullableBool has a value: {nullableBool.HasValue}");

      // Default value assignment for nullable DateTime
      DateTime defaultDateTime = nullableDateTime ?? DateTime.MinValue;
      Console.WriteLine($"defaultDateTime: {defaultDateTime}"); // Output: defaultDateTime:
1/1/0001 12:00:00 AM
```

```
    }
}
```

Benefits of Nullable Types

1. **Nullability**: You can represent missing or undefined values for value types, which isn't possible with non-nullable value types.
2. **Database Handling**: Nullable types are useful when dealing with databases where fields can have NULL values.
3. **Flexible Assignments**: Nullable types allow better handling of cases where a value might not always be available.

Important Notes

- Accessing the Value property of a nullable type when it is null will result in a InvalidOperationException.
- Use the HasValue property to safely check if a nullable type has a value before accessing its Value property.

Common Scenarios for Using Nullable Types

1. **Database Fields**: When mapping database columns that allow NULL values to application variables, nullable types come in handy.
2. **Optional Parameters**: Nullable types can be used for parameters that might not always be provided.
3. **Error Handling**: Nullable types allow you to represent the absence of a value as a valid scenario (e.g., null for missing data).

var vs dynamic in C#

In C#, var and dynamic are both used to handle variables, but they have distinct differences in how type determination and behavior work. Here's a detailed comparison between var and dynamic:

Feature	var	dynamic
Type Determination	Compile-time	Runtime
Type Safety	Strong (compile-time checking)	Weak (runtime checking)
Initialization	Required at declaration	Not required at declaration
Type Change	Not allowed (once declared)	Allowed (can change type at runtime)
Performance	Faster (no runtime resolution)	Slower (runtime resolution needed)

Feature	var	dynamic
Class-Level Fields	Not allowed	Allowed
Use Case	When type is known or can be inferred	When type can vary or is unknown at compile-time

Type Determination:

var: The type of a variable declared with var is determined at compile-time. It must be initialized with a value so that the compiler can infer the type.
Example:
var number = 10; // Compiler infers type as 'int'

dynamic: The type of a variable declared with dynamic is determined at runtime. The compiler does not check for type correctness, so you can assign any type to a dynamic variable during runtime.
Example:
dynamic obj = 10; // Initially an 'int'
obj = "Hello"; // Later becomes a 'string'

Type Safety:

var: Offers **strong type safety**. The type is checked at compile-time, and any type mismatches will cause compile-time errors.
Example:
var num = 5; // type is inferred as 'int'
num = "Hello"; // Error: Cannot assign string to int

dynamic: Has **weak type safety**. The type is not checked at compile-time. If you perform invalid operations or type mismatches, they will only be detected at runtime.
Example:
dynamic obj = 10; // Initially an 'int'
obj = "Hello"; // Allowed, no compile-time error
obj = obj + 5; // Runtime error: Cannot add a string and int

Initialization:

var: Must be initialized at the time of declaration. You cannot declare a var variable without an initializer because the compiler needs to infer the type.
Example:
var x; // Error: Implicitly typed variables must be initialized

dynamic: Can be declared without initialization. You can assign a value to it later, and its type will be determined at runtime.
Example:
dynamic y; // No initialization required

```
y = 10;    // Type determined as int at runtime
```

Type Change:

var: Once a variable is declared with var, its type cannot be changed. The type is fixed and cannot be reassigned to a different type.
Example:

```
var x = 5;  // 'x' is an 'int'
x = "hello"; // Error: Cannot assign string to int
```

dynamic: The type of a dynamic variable can change at runtime. You can assign a value of any type to a dynamic variable after it has been declared.
Example:

```
dynamic y = 10;  // 'y' is an 'int'
y = "hello";    // Now 'y' is a string
```

Performance:

var: Since var is evaluated at compile-time, it does not incur the overhead of runtime type checking. Therefore, it is faster compared to dynamic.
Example:

```
var x = 5;  // No runtime resolution, faster
```

dynamic: Since dynamic requires runtime resolution of types and method invocations, it incurs slower performance due to the need for runtime type checking.
Example:

```
dynamic y = 5;
y = y + 10;   // Slower due to runtime resolution of types
```

Class-Level Fields:

var: Cannot be used for class-level fields or members. It is only used for local variable declarations.
Example:

```
public class MyClass
{
   var x = 10;  // Error: 'var' cannot be used for class fields
}
```

dynamic: Can be used for class-level fields or members, as it is not limited to local variables.
Example:

```
public class MyClass
{
   public dynamic x = 10;  // Valid
}
```

⍰ Use var when the type is known and can be inferred at compile-time, ensuring type safety and better performance.

⍰ Use dynamic when the type cannot be determined at compile-time or is likely to change at runtime, but be aware of the potential performance costs and lack of compile-time type safety.

References

- https://dotnet.microsoft.com/en-us/
- https://visualstudio.microsoft.com/free-developer-offers/
- https://learn.microsoft.com/en-gb/dotnet/fundamentals/
- https://learn.microsoft.com/en-gb/dotnet/core/introduction?WT.mc_id=dotnet-35129-website
- https://learn.microsoft.com/en-gb/training/dotnet/?WT.mc_id=dotnet-35129-website
- https://en.wikipedia.org/wiki/.NET_Framework
- https://en.wikipedia.org/wiki/.NET_Framework_version_history
- https://en.wikipedia.org/wiki/.NET

Assignment

- https://learn.microsoft.com/en-gb/training/modules/csharp-write-first/
- https://learn.microsoft.com/en-gb/training/modules/dotnet-introduction/
- https://learn.microsoft.com/en-gb/training/modules/csharp-write-first/1-introduction
- https://learn.microsoft.com/en-gb/training/modules/csharp-literals-variables/
- https://learn.microsoft.com/en-gb/training/modules/csharp-basic-formatting/
- https://learn.microsoft.com/en-gb/training/modules/csharp-basic-operations/
- https://learn.microsoft.com/en-gb/training/modules/guided-project-calculate-print-student-grades/
- https://learn.microsoft.com/en-gb/training/modules/guided-project-calculate-final-gpa/

- https://learn.microsoft.com/en-us/training/paths/get-started-c-sharp-part-1/

- https://learn.microsoft.com/en-us/training/paths/get-started-c-sharp-part-2/
- https://learn.microsoft.com/en-us/training/paths/get-started-c-sharp-part-3/
- https://learn.microsoft.com/en-us/training/paths/get-started-c-sharp-part-4/
- https://learn.microsoft.com/en-us/training/paths/get-started-c-sharp-part-5/
- https://learn.microsoft.com/en-us/training/paths/get-started-c-sharp-part-6/

- Given an array of binary digits, 0 and 1, sort the array so that all zeros are at one end and all ones are at the other. Which end does not matter. Determine the minimum number of swaps to sort the array.
  Example:arr=[0,1,0,1,0]
  With 1 move, switching elements 1 and 4, yields [0,0,0,1,1], a sorted array.

- Provide some Array members and sample examples
  Properties: Length, Rank
  Methods: Clear, Copy, CopyTo, IndexOf, LastIndexOf, Reverse, Sort, BinarySearch

```
using System;

public class ArrayExample
{
   public static void Main()
   {
      int[] numbers = { 5, 2, 9, 1, 5, 6 };

      // Length
      Console.WriteLine("Length: " + numbers.Length);

      // Rank
      Console.WriteLine("Rank: " + numbers.Rank);

      // IndexOf
      int index = Array.IndexOf(numbers, 9);
      Console.WriteLine("Index of 9: " + index);

      // Sort
      Array.Sort(numbers);
      Console.WriteLine("Sorted array: " + string.Join(", ", numbers));

      // Reverse
      Array.Reverse(numbers);
      Console.WriteLine("Reversed array: " + string.Join(", ", numbers));
```

```
        // Copy
        int[] copy = new int[numbers.Length];
        Array.Copy(numbers, copy, numbers.Length);
        Console.WriteLine("Copied array: " + string.Join(", ", copy));

        // Clear
        Array.Clear(copy, 0, copy.Length);
        Console.WriteLine("Cleared array: " + string.Join(", ", copy));
    }
}
```

- write a program of fibonacci series

```
using System;

public class FibonacciSeries
{
    public static void Main()
    {
        Console.Write("Enter the number of terms for the Fibonacci series: ");
        int terms = int.Parse(Console.ReadLine());

        // Display the Fibonacci series
        PrintFibonacciSeries(terms);
    }

    public static void PrintFibonacciSeries(int terms)
    {
        int first = 0, second = 1, next;

        Console.WriteLine("Fibonacci Series:");

        for (int i = 0; i < terms; i++)
        {
            if (i == 0)
            {
                Console.Write(first + " ");
                continue;
            }
            if (i == 1)
            {
                Console.Write(second + " ");
                continue;
            }
            next = first + second;
            first = second;
            second = next;
```

```
        Console.Write(next + " ");
      }
    }
}
```

- Write a function that takes an integer i and returns a string with the integer backwards followed by the original integer.
  123 ➞ "321123" (Coding question)

```
using System;

public class ReverseAndAppend
{
    public static void Main()
    {
        int number = 123;
        string result = ReverseAndAppendInteger(number);
        Console.WriteLine(result); // Output: 321123
    }

    public static string ReverseAndAppendInteger(int i)
    {
        // Convert the integer to a string
        string original = i.ToString();

        // Reverse the string
        char[] charArray = original.ToCharArray();
        Array.Reverse(charArray);
        string reversed = new string(charArray);

        // Append the original string to the reversed string
        return reversed + original;
    }
}
```

- Provide some string members and sample examples
  Properties: Length,
  Methods: IndexOf, LastIndexOf, Contains, Replace, ToUpper and ToLower, Trim, TrimStart, TrimEnd, Split, Join,

```
using System;

public class StringExample
{
```

```
    public static void Main()
    {
        string text = "   Hello, World!   ";

        // Length
        Console.WriteLine($"Length: {text.Length}");

        // Substring
        string sub = text.Substring(7, 5);
        Console.WriteLine($"Substring: {sub}");

        // IndexOf
        int index = text.IndexOf("World");
        Console.WriteLine($"Index of 'World': {index}");

        // Contains
        bool contains = text.Contains("Hello");
        Console.WriteLine($"Contains 'Hello': {contains}");

        // Replace
        string replaced = text.Replace("World", "C#");
        Console.WriteLine($"Replaced: {replaced}");

        // ToUpper and ToLower
        string upper = text.ToUpper();
        string lower = text.ToLower();
        Console.WriteLine($"Uppercase: {upper}");
        Console.WriteLine($"Lowercase: {lower}");

        // Trim
        string trimmed = text.Trim();
        Console.WriteLine($"Trimmed: '{trimmed}'");

        // Split
        string[] words = text.Split(',');
        Console.WriteLine("Split:");
        foreach (string word in words)
        {
            Console.WriteLine(word);
        }

        // Join
        string joined = string.Join(", ", words);
        Console.WriteLine($"Joined: {joined}");
    }
}
```

- Reverse a string
- Find the second smallest element in unsorted array. You can't sort it.
- Given an interger array of size n filled with zero and ones ,
  Arrange all zeros on left and all one on right side.
  int a[n] = {01010101}
- Write a C# program to get all possible substrings from a string input
  input: abcd
  output: a ab abc abcd b bc bcd c cd d
- Provide Datetime members with examples
- Provide Object class members with examples
- Try different data conversion scenarios (Tostring(), string to numeric types)
- Is, as operators

Self-Check

- What is Type Safe?
- what and how JIT compilation works
- What is CLR, CLS , CTS, CLI, BCL ?
- What is String Interpolation?
- What is string verbatim?
- Data Conversions
- What are value and reference types
- What is the difference between stack and Heap memory
- how to return multiple output from function
- What is the difference between string and string builder class
- What is boxing and unboxing
- var vs dynamic
- What are the main components of the .NET Framework?
- Explain the role of CLR in .NET Framework.
- What is JIT compilation? How does it work in .NET?
- What is the difference between value types and reference types in .NET?
- What is the purpose of namespaces in .NET?
- Explain the concept of assemblies in .NET.
- What are the different types of assemblies in .NET?
- What is the Global Assembly Cache (GAC)?
- What is the role of the Base Class Library (BCL) in .NET?
- What are the different types of application domains in .NET?
- What is the significance of the Main method in a C# application?
- Explain the difference between managed and unmanaged code.
- What is boxing and unboxing in .NET?
- What are the components of CLR?
- What is the difference between double and decimal?
- What are jagged arrays?