**C# + OOPS Based**

What is assembly in C#? Explain types of assemblies.

“Collection of resources & types that are designed to work together as a logical unit of functionality”

Assemblies can either be DLL (dynamic link libraries) or .exe files depending on their intended use. They provide the common language runtime with the information it needs to be aware of type implementations.

1. Assemblies are building blocks of the .NET applications and play a crucial role in development by providing a way to package and deploy code.
2. In a .NET framework, we can build assembly from one or more source code files. Assemblies can contain one or more modules.
3. Assemblies are only loaded into memory if they are required and thus assemblies can be an efficient way to manage resources in larger projects.
4. For libraries that target .NET Framework, you can share assemblies between applications by putting them in the global assembly cache (GAC). GAC is a machine wide cache used to store assemblies that are shared by multiple applications and thus the assemblies in the GAC should be strongly named which ensures their uniqueness.

Types of Assemblies

--------------------

**Private Assembly**: An Assembly that is solely used by one application is referred to as a Private Assembly. It is typically found in the directory for the application or a subdirectory of the directory for the program. Private Assemblies are not intended to be shared with other applications. They are used to store application-specific code and resources. Private Assemblies are created when an application is compiled. When an application is compiled, all the code files and resources that are used by the application are compiled into a single assembly.

Private Assemblies are simple to deploy and use. They don't need any extra installation or setting. They are automatically loaded by the .NET runtime when the application starts.

**Shared Assemblies**: An assembly that is used by several programs is referred to as a shared assembly. It is typically found in the **Global Assembly Cache (GAC)** or a common directory. Multiple applications are supposed to share a shared assembly. They are used to store resources and code that are shared by various applications.

Shared Assemblies are created using the strong name tool ***(sn.exe)***. A digital signature for the assembly is applied using the strong name tool. The digital signature guarantees that the assembly is genuine and unaltered.

The **Global Assembly Cache (GAC)** houses shared assemblies. Shared assemblies are kept in the **GAC**, which serves as a central repository. The **GAC** location is in the Windows directory ***(C:\Windows\assembly)***. Shared assemblies are registered with the **GAC** using the gacutil.exe tool.

Shared assemblies require special configuration and installation. They cannot be simply copied to the application's directory. They need to be registered with the **GAC** using the ***gacutil.exe*** tool.

**Satellite Assemblies**: An assembly used to store regional resources is referred to as a Satellite Assembly. It is typically found in a subdirectory of the directory for the application or a subdirectory of the directory for the Shared Assembly. Localized resources like strings, pictures, and audio files are kept in satellite assemblies.

Satellite Assemblies are created using the ***resgen.exe*** tool. The ***resgen.exe*** tool is used to create a resource file ***(.resx)*** from a text file. The resource file is then compiled into a satellite assembly using the ***al.exe*** tool.

Satellite Assemblies are named using a specific naming convention. The naming convention for satellite assemblies is as follows:

* <AssemblyName>.resources.dll

The **<AssemblyName>** part of the name is the name of the main assembly that the satellite assembly is associated with. The.NET runtime automatically loads satellite assemblies when the application first launches. The .NET runtime automatically selects the appropriate satellite assembly based on the user's culture settings.

**Dynamic Link Libraries (DLLs):** A **Dynamic Link Library (DLL)** is a type of assembly that contains code that can be used by multiple applications. Similar to shared assemblies, **DLLs** are created with the intention of being used by numerous applications. **DLLs**, however, differ from shared assemblies in that they do not have a distinctive name.

**DLLs** are kept in the directory of the application or a subdirectory of the directory of the application. They can be used by multiple applications by simply copying the **DLL** to the application's directory.

**DLLs** are created using the same tools that are used to create shared assemblies. However, **DLLs** do not require a strong name. They can be signed with a digital signature, but this is optional.

------------------

What is garbage collection in C#? How does the garbage collector determine which objects are eligible for garbage collection?

**Garbage Collection (GC)** in C# is the process of automatically reclaiming memory occupied by objects that are no longer in use by the application. The GC is part of the **Common Language Runtime (CLR)** and helps manage the application's memory by freeing up space occupied by unused objects, thus preventing memory leaks.

**How GC Works:** The **Garbage Collector** works by:

1. Identifying objects that are no longer referenced by any part of the application.
2. Releasing the memory occupied by these unreferenced objects.
3. Compacting the memory to make space for new objects.

**Eligibility for Garbage Collection:**

An object becomes eligible for garbage collection when it is **no longer reachable** from any part of the code. The GC starts from **roots** (like global variables, local variables in active methods, and static variables) and checks all objects that can be reached directly or indirectly. Any object that cannot be reached is considered for garbage collection. **Example:**

class Program{

static void Main() {

Employee emp = new Employee(); // emp is a reference

emp = null; // Now the Employee object is eligible for garbage collection

}

}

NOTE: when emp is set to null, the reference to the Employee object is lost, making it eligible for garbage collection.

**Generations in Garbage Collection:**

The .NET GC uses a generational approach:

* **Generation 0**: Newly created objects. Short-lived objects are usually collected here.
* **Generation 1**: Objects that survived Generation 0 collections.
* **Generation 2**: Long-lived objects, such as static objects or global data.

----------

What is operator overloading?

list the operator that can be and that cannot be overloaded.

Operator overloading is a technique for implementing polymorphism, which is an object-oriented programming paradigm. Polymorphism allows for multiple interchangeable classes that implement methods and properties differently.

**Operator Overloading** in C# allows you to define or change the behavior of operators (like +, -, \*, etc.) for user-defined types (e.g., classes or structs). This enables operators to work with objects just like they do with primitive types. Essentially, it provides a way to specify how operators should function for your own types. Example:

class Complex {

public double Real { get; set; }

public double Imaginary { get; set; }

// Constructor

public Complex(double real, double imaginary)

{

Real = real;

Imaginary = imaginary;

}

// Overload the + operator

public static Complex operator +(Complex c1, Complex c2)

{

return new Complex(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);

}

public override string ToString()

{

return $"{Real} + {Imaginary}i";

}

}

class Program

{

static void Main()

{

Complex c1 = new Complex(1.5, 2.5);

Complex c2 = new Complex(3.0, 1.0);

Complex sum = c1 + c2; // Using the overloaded + operator

Console.WriteLine(sum); // Output: 4.5 + 3.5i

}

}

Operators That Cannot Be Overloaded:

* + Conditional Logical Operators: && (logical AND), || (logical OR)
  + Object Creation and Management: new, sizeof, typeof
  + Equality Checks: = (assignment), +=, -= (compound assignment operators can't be overloaded directly, though they use other operators that can be overloaded)
  + Reference and Dereference Operators: . (member access), ?: (ternary conditional), => (lambda expression), is, as, checked, unchecked
  + Type Casting: typeof, await

Rules for Operator Overloading:

1. **Only existing operators can be overloaded**: You cannot create new operators.
2. **At least one operand must be a user-defined type**: You can't overload operators solely for built-in types.
3. **Operators are defined as static methods**: The method defines the behavior of the operator.
4. **Certain pairs of operators must be overloaded together**: For example, if you overload the == operator, you must also overload the != operator.

---------------------------------------

**.NET Based**

**Define CLS and CTS in .NET Framework.**

The .NET Framework uses two important concepts to ensure cross-language compatibility: CLS (Common Language Specification) and CTS (Common Type System).

**CLS:**  stands for Common Language Specification and it is a subset of CTS. It defines a set of rules and restrictions that every language must follow which runs under the .NET framework. The languages which follow this set of rules are said to be CLS Compliant. In simple words, CLS enables cross-language integration.

The CLS is a specification that defines the rules for supporting the language integration in a certain way that the programs are written in any language, still, it can interoperate with the one another seamlessly while taking the full advantage of concepts such as exceptions handling, inheritance, polymorphism, and other features accordingly. If you write code that only uses CLS-compliant features, your code can be used by other .NET languages.

**For example**: one rule is that you cannot use multiple inheritances within .NET Framework. As you know C++ supports multiple inheritances but; when you will try to use that C++ code within C#, it is not possible because C# doesn’t support multiple inheritances.

**CTS:** In .NET, every Data Type is internally represented by a class or structure. All the classes and structures related to Data Types are collectively known as CTS. As you know every language provides its own keywords for Data Types but internally all the languages which run under the .NET framework use the classes and structures available in CTS.

**For example**, C# has int Data Type and VB.Net has Integer Data Type. Hence a variable is declared as an int in C# or Integer in vb.net, finally, after compilation, use the same structure Int32 from CTS.

All the structures and classes available in CTS are common for all .NET Languages and the purpose of these is to support language independence in .NET. Hence it is called CTS.

The **CTS** defines how data types should be declared, used, and managed in the .NET Framework.

Categories of CTS Types:

1. **Value Types**: Directly contain data (e.g., int, float).
2. **Reference Types**: Store references to the data (e.g., object, class).

**Example**: In C#, both int and System.Int32 are CTS-compliant and can be used interchangeably.

int a = 10; // int is an alias for System.Int32

System.Int32 b = 20;

**What is the use of Microsoft Intermediate Language (MSIL) and JIT Compiler in the .NET Framework?**

Microsoft Intermediate Language (MSIL):

* When you compile a C# program, the source code is not directly converted into machine code. Instead, it is compiled into MSIL, a platform-independent set of instructions.
* MSIL is a low-level, CPU-independent instruction set that can be converted into native machine code for different platforms.
* MSIL is stored in assemblies (.exe or .dll files) and can be executed on any platform that supports the .NET runtime.

In the .NET Framework, Microsoft Intermediate Language (MSIL) acts as a platform-independent bytecode that is generated when you compile code written in any .NET language, allowing for cross-language compatibility, while the Just-In-Time (JIT) compiler then translates this MSIL into machine code specific to the target computer at runtime, optimizing performance for the current system; essentially, MSIL provides a common language for different .NET languages to communicate, and the JIT compiler converts it into executable code on the fly as needed.

The Just-In-Time (JIT) compiler in the .NET framework converts Intermediate Language (IL) code into native machine code that can be executed by the computer's hardware. The JIT compiler has several uses, including:

* Optimizing performance: The JIT compiler can optimize the most used IL code for faster performance.
* Making applications portable: The JIT compiler's two-step process makes it easier to make applications work on different platforms.
* Caching native code: The JIT compiler caches native code so that it can be reused when the same method is called again.

**Just-In-Time (JIT) Compiler:**

* At runtime, the **JIT Compiler** converts the MSIL code into **native machine code** that is specific to the operating system and hardware on which the application is running.
* JIT compilation happens **just before execution** and allows the .NET Framework to provide platform independence.
* .NET uses **three types of JIT**:
  1. **Pre-JIT**: Compiles the entire code into machine code in a single process (done during deployment).
  2. **Econo-JIT**: Compiles code on-demand but discards it when not needed.
  3. **Normal-JIT**: Compiles code on-demand and keeps it in memory for future use.

DART

What are Getters and Setters? Write the syntax to define them in Dart.

**Getters and Setters** methods are the class methods used to manipulate the data of the class fields. Getter is used to read or get the data of the class field whereas setter is used to set the data of the class field to some variable. They allow controlled access to private properties of a class. They provide encapsulation by allowing read and write access to class properties while ensuring that some constraints or checks can be applied before accessing or modifying them.

* **Getter**: Used to retrieve the value of a private field.
* **Setter**: Used to update the value of a private field with validation logic if needed.

(code example copy + keyword example + explain code)

Why does Dart not use destructor function?

Dart does not support explicit destructors like in languages such as C++ because **Dart uses garbage collection** for memory management. In garbage-collected languages like Dart, the system automatically reclaims memory when objects are no longer in use, so there’s no need for the developer to manually handle object destruction.

* The garbage collector automatically deallocates objects when they are no longer referenced.
* **Finalizers** can be used if specific cleanup is needed, but Dart avoids manual destructors to simplify memory management.

What is operator overloading? Give an example of binary operator overloading.

**Operator Overloading** allows a class to redefine or overload existing operators to provide custom behavior when they are used with instances of that class. In Dart, you can overload operators like +, -, \*, etc., by defining special methods in the class.

Example:

class Point {

int x, y;

Point(this.x, this.y);

// Overloading the + operator

Point operator +(Point other) {

return Point(this.x + other.x, this.y + other.y);

}

}

void main() {

Point p1 = Point(2, 3);

Point p2 = Point(4, 5);

Point p3 = p1 + p2; // Using the overloaded + operator

print('x: ${p3.x}, y: ${p3.y}'); // Output: x: 6, y: 8

}

* The operator keyword is used to define a custom behavior for the + operator.
* In the example, two Point objects are added together by overloading the + operator.

--------------------

Write about the Dart static keyword and Dart static variable

**Definition:** In Dart, the static keyword is used to declare class-level variables and methods. These variables and methods belong to the class rather than instances of the class, meaning they can be accessed without creating an object of the class.

**Key Points:**

* **Static Variables**: Shared by all instances of the class. They are initialized once and retain their value between instances.
* **Static Methods**: Can only access other static members (variables or methods) and cannot access instance variables directly.

class Example {

static int count = 0;

static void incrementCount() {

count++;

}

void showCount() {

print("Count: $count");

}

}

void main() {

Example.incrementCount(); // Accessing static method without instance

Example.incrementCount();

Example example1 = Example();

example1.showCount(); // Output: Count: 2

}

* The count variable and incrementCount() method are marked as static, meaning they belong to the class Example itself.
* The static method incrementCount() can be called without creating an instance of the Example class, and it increments the shared count variable.
* Any instance of Example can access and display the static variable count.

---------------------------

Define abstract classes in Dart programming. How to declare the abstract class? Write the syntax

**Definition:** An abstract class in Dart is a class that cannot be instantiated directly. It serves as a blueprint for other classes. Abstract classes are used to define methods that must be implemented by subclasses. To declare an abstract class, we use the **abstract**keyword.

Abstract classes can have both abstract methods (without implementation) and regular methods (with implementation).

It’s important to note that a class declared as abstract may or may not include abstract methods. However, if a class includes an abstract method, it must be declared as abstract.

Features of Abstract Class:

* **Abstract Methods**: A class containing an abstract method must be declared abstract. An abstract class may contain both abstract and concrete methods.
* **Declaration**: A class can be declared abstract by using the abstract keyword.
* **Initialization**: An abstract class cannot be instantiated.
* **Inheritance**: An abstract class can be extended, but any subclass must implement all abstract methods.

(Code example of abstraction)