1. **Core Concepts (C# 1.0 to 2.0)**

Introduction to .NET and C#

- **Question:** What is the .NET Framework, and how does the CLR work?
  Answer: The .NET Framework is a platform for building and running Windows applications. CLR (Common Language Runtime) manages the execution of .NET programs, offering memory management, type safety, exception handling, and garbage collection.
- **Question:** What are value types and reference types in C#?
  Answer: Value types are stored in the stack and hold actual data (e.g., `int`, `double`), while reference types are stored in the heap and hold a reference to the actual data (e.g., `class`, `string`).

Data Types and Variables

- **Question:** What are nullable types in C#, and why are they important?
  Answer: Nullable types allow value types (e.g., `int`, `bool`) to represent null values. This is important for scenarios like database interaction where a field may or may not have a value.

Classes, Objects, and Structs

- **Question:** What is the difference between a class and a struct?
  Answer: A class is a reference type, and its instances are allocated on the heap, while a struct is a value type, and its instances are allocated on the stack.

Methods and Parameters

- **Question:** What are the differences between `ref` and `out` parameters in C#?
  Answer: `ref` requires that a variable be initialized before it is passed, while `out` allows a method to initialize a parameter and pass it back to the caller.

---

2. **Object-Oriented Programming (OOP)**

Encapsulation, Inheritance, and Polymorphism

- **Question:** How does encapsulation protect data in C#?
  Answer: Encapsulation restricts access to certain data by using access modifiers like `private`, `protected`, and `public`, allowing only authorized methods to modify it.

Abstract Classes and Interfaces

- **Question:** What is the difference between an abstract class and an interface?
  Answer: An abstract class can contain method implementations and fields, while an interface can only declare methods and properties. Classes can inherit multiple interfaces but only one abstract class.

Overloading and Overriding

- **Question:** What is the difference between method overloading and method overriding?
  Answer: Overloading allows multiple methods in the same class to have the same name but different signatures. Overriding allows a subclass to provide a specific implementation for a method defined in a base class using the `override` keyword.

---

3. **Error Handling and Exceptions**

Exception Hierarchy

- **Question:** What is the base class for all exceptions in C#?
  Answer: The base class for all exceptions in C# is `System.Exception`.

Try-Catch-Finally Blocks

- **Question:** What is the purpose of the `finally` block in exception handling?
  Answer: The `finally` block is used to execute code after the try/catch, regardless of whether an exception occurred, typically for resource cleanup like closing file handles or database connections.

Custom Exceptions

- **Question:** How do you create a custom exception in C#?
  Answer: You create a custom exception by deriving a new class from `System.Exception` and implementing necessary constructors.

```csharp
public class CustomException : Exception { public CustomException(string message) : base(message) { } }
```

---

4. **Delegates and Events**

Delegates in C#

- **Question:** What are delegates, and how are they used?
  Answer: Delegates are type-safe pointers to methods. They allow methods to be passed as parameters. For example:

```csharp
public delegate void PrintDelegate(string message); public void PrintMessage(PrintDelegate print) { print("Hello!"); }
```

Event Handling in C#

- **Question:** How do events and delegates work together in C#?
  Answer: Events use delegates to notify subscribers when something occurs. For example:

```csharp
public event PrintDelegate PrintEvent; PrintEvent?.Invoke("Hello World");
```

---

5. **Collections and Generics (C# 2.0)**

Generic Collections

- **Question:** What is the benefit of using generic collections in C#?
  Answer: Generic collections like `List<T>` and `Dictionary<K,V>` are type-safe and avoid boxing/unboxing, providing better performance and fewer runtime errors compared to non-generic collections.

Covariance and Contravariance in Generics

- **Question:** What is covariance and contravariance in C# generics?
  Answer: Covariance allows a method to return a more derived type than specified by the generic parameter, while contravariance allows a method to accept a parameter of a less derived type.

---

6. **C# 3.0 Features**

Anonymous Methods and Lambda Expressions

- **Question:** What is the difference between an anonymous method and a lambda expression?
  Answer: Anonymous methods and lambda expressions both allow you to define a method inline, but lambda expressions are more concise and are the preferred syntax.

  ```csharp
  Func<int, int> square = x => x * x;
  ```

LINQ

- **Question:** What is LINQ, and why is it useful?
  Answer: LINQ (Language Integrated Query) allows querying collections in a consistent way using query syntax or method syntax. It simplifies querying by integrating directly into C#.

---

7. **C# 4.0 Features**

Dynamic Programming

- **Question:** What is the `dynamic` keyword in C#, and when would you use it?
  Answer: The `dynamic` keyword allows for late binding, meaning the type is resolved at runtime rather than compile time. It's useful for working with COM objects, reflection, or interop scenarios.

Optional and Named Parameters

- **Question:** What are named and optional parameters in C#?
  Answer: Named parameters allow you to specify arguments by parameter name, and optional parameters let you define default values for function parameters.

---

8. **Asynchronous Programming (C# 5.0)**

Async and Await Keywords

- **Question:** How do `async` and `await` work in C#?
  Answer: `async` marks a method as asynchronous, and `await` is used to pause execution of the method until the awaited task completes, allowing other code to run in the meantime.

Exception Handling in Asynchronous Methods

- **Question:** How is exception handling different in asynchronous methods?
  Answer: Exceptions in asynchronous methods are thrown as part of the returned `Task`, and they need to be caught using `try-catch` blocks or checked via the `Task.Exception` property.

---

9. **C# 6.0 Features**

Expression-Bodied Members

- **Question:** What are expression-bodied members in C#?
  Answer: Expression-bodied members allow methods, properties, or constructors to be written in a more concise syntax using the `=>` operator. For example:

  ```csharp
  public string Name => "John";
  ```

String Interpolation

- **Question:** How does string interpolation work in C#?
  Answer: String interpolation allows you to embed expressions directly within a string, prefixed with `$`:

  ```csharp
  string name = "John"; Console.WriteLine($"Hello, {name}");
  ```

---

10. **C# 8.0 Features**

Nullable Reference Types

- **Question:** What are nullable reference types in C#?
  Answer: Nullable reference types enable you to distinguish between nullable and non-nullable reference types, reducing the likelihood of null reference exceptions.

Asynchronous Streams

- **Question:** What are asynchronous streams, and how do they work in C#?
  Answer: Asynchronous streams allow you to asynchronously iterate over a collection of data using `IAsyncEnumerable<T>` and `await foreach`.

---

11. **Memory Management and Performance**

Garbage Collection (GC)

- **Question:** How does the garbage collector work in .NET?
  Answer: The garbage collector (GC) automatically manages memory allocation and deallocation by identifying and freeing objects that are no longer in use.

---

12. **Multithreading and Parallel Programming**

Task Parallel Library (TPL)

- **Question:** What is the Task Parallel Library (TPL) in C#?
  Answer: TPL provides a set of APIs for parallel programming, allowing you to create and

manage tasks that run asynchronously or concurrently using `Task.Run()`, `Task.WaitAll()`, and other methods.

---

13. **C# 9.0 Features**

Record Types

- **Question:** What are record types in C# 9.0?
  Answer: Record types in C# are reference types that provide built-in value semantics for equality and immutability. They are often used for immutable data models.

```csharp
public record Person(string FirstName, string LastName);
```

Init-Only Setters

- **Question:** What is the purpose of `init` in C# 9.0?
  Answer: `init` allows properties to be set during object initialization but prevents them from being modified afterward, providing immutability after construction.

```csharp
public string Name { get; init; }
```

---

14. **C# 10.0 Features**

Global Usings

- **Question:** What are global usings in C# 10.0?
  Answer: Global usings allow you to define `using` directives at a global level, meaning they are applied across all files in a project, reducing redundancy.

```csharp
global using System;
```

File-Scoped Namespaces

- **Question:** What are file-scoped namespaces in C# 10.0?
  Answer: File-scoped namespaces allow you to define a namespace that applies to an entire file without wrapping the entire file in curly braces, leading to cleaner code:

```csharp
namespace MyNamespace;
```

---

15. C# 11.0 Features

Raw String Literals

- **Question:** What are raw string literals in C# 11.0?
  Answer: Raw string literals allow you to define strings with multi-line content and without needing to escape characters like quotes or backslashes. This is useful for JSON or HTML strings.

```csharp
string json = """ { "name": "John" } """;
```

List Patterns

- **Question:** How do list patterns work in C# 11.0?
  Answer: List patterns allow you to match lists against specific patterns, making it easier to work with arrays and collections. For example:

```csharp
if (list is [1, 2, 3]) { Console.WriteLine("Matched!"); }
```

---

16. C# 12.0 Features

Primary Constructors for Classes

- **Question:** What are primary constructors in C# 12.0?
  Answer: Primary constructors allow you to define a constructor directly within the class declaration, simplifying the initialization process:

```csharp
class Person(string name, int age);
```

Required Members

- **Question:** What are required members in C# 12.0?
  Answer: Required members enforce that specific properties must be set during object initialization, providing more control over object construction:

```csharp
public required string Name { get; set; }
```

---

17. Memory Management and Performance

Boxing and Unboxing

- **Question:** What is boxing and unboxing in C#?
  Answer: Boxing is the process of converting a value type to a reference type (e.g., `int` to `object`). Unboxing is the reverse process. Boxing incurs a performance penalty, so it's generally avoided when possible.

```csharp
object obj = 42; // Boxing int num = (int)obj; // Unboxing
```

Span<T> and Memory<T>

- **Question:** How do `Span<T>` and `Memory<T>` improve performance in C#?
  Answer: `Span<T>` and `Memory<T>` provide a way to work with memory efficiently without creating new arrays or allocating on the heap, reducing the need for garbage collection.

```csharp
Span<int> span = stackalloc int[5];
```

---

18. Reflection and Metadata

Reflection API

- **Question:** What is reflection in C#, and how is it used?
  Answer: Reflection allows inspection of assemblies, types, and members at runtime. It is commonly used for dynamically loading assemblies, accessing methods, and retrieving metadata.

```csharp
Type type = typeof(Person); MethodInfo method = type.GetMethod("SayHello");
```

Custom Attributes

- **Question:** How do you create and use custom attributes in C#?
  Answer: Custom attributes allow developers to annotate code with metadata. You create a custom attribute by inheriting from `System.Attribute` and can apply it to classes, methods, or properties.

```csharp
[AttributeUsage(AttributeTargets.Class)] public class MyCustomAttribute : Attribute
{ }
```

---

19. Multithreading and Parallel Programming

Task Parallel Library (TPL)

- **Question:** What is the purpose of the Task Parallel Library (TPL) in C#?
  Answer: TPL simplifies the process of parallel programming by using tasks that run concurrently or asynchronously. It provides abstractions like `Task`, `Parallel.For`, and `Parallel.ForEach`.

```csharp
```

```
Task.Run(() => DoWork());
```

Synchronization (Locks, Mutex, Semaphore)

- **Question:** How do you implement thread synchronization in C#?
  Answer: Synchronization ensures that multiple threads don't access shared resources simultaneously. This can be achieved using `lock`, `Mutex`, `Semaphore`, or other synchronization mechanisms.

```csharp
lock(lockObject) { // Critical section }
```

---

20. **Dependency Injection (DI) and Inversion of Control (IoC)**

DI Patterns and Frameworks in .NET

- **Question:** How does dependency injection work in .NET, and why is it important?
  Answer: Dependency injection (DI) decouples class dependencies by injecting them via constructors or methods, improving modularity and testability. In .NET Core, DI is built-in via `IServiceCollection`.

```csharp
services.AddTransient<IService, ServiceImplementation>();
```

Service Lifetimes

- **Question:** What are the different service lifetimes in DI?
  Answer: The three lifetimes are:
  - **Transient:** Created every time requested.
  - **Scoped:** Created once per request.
  - **Singleton:** Created once and reused throughout the application.

---

21. **Design Patterns in C#**

Singleton Pattern

- **Question:** What is the Singleton pattern, and how is it implemented in C#?
  Answer: The Singleton pattern ensures that only one instance of a class is created. This is useful for resources like logging or configuration. It is implemented by making the constructor private and providing a static instance.

```csharp
public class Singleton { private static readonly Singleton instance = new Singleton(); private Singleton() { } public static Singleton Instance => instance; }
```

Factory Pattern

- **Question:** How does the Factory pattern work in C#?
  Answer: The Factory pattern provides a way to create objects without specifying the exact class

of the object that will be created. It is useful when the object creation process is complex or involves multiple steps.

```csharp
public interface IProduct { } public class ProductA : IProduct { } public class ProductFactory { public IProduct CreateProduct() => new ProductA(); }
```

---

22. **Testing in C#**

Unit Testing with MSTest, NUnit, or xUnit

- **Question:** What is unit testing, and how do you perform it in C#?
  Answer: Unit testing involves testing individual components of code in isolation to ensure they work as expected. Tools like MSTest, NUnit, and xUnit provide frameworks for writing and running unit tests.

```csharp
[TestMethod] public void TestAddMethod() { var result = calculator.Add(2, 3); Assert.AreEqual(5, result); }
```

Mocking Frameworks

- **Question:** What is mocking in unit testing, and how is it used in C#?
  Answer: Mocking is the process of simulating the behavior of dependencies in a class, allowing you to isolate the unit of work being tested. Popular mocking frameworks include Moq and NSubstitute.

```csharp
var mockService = new Mock<IService>(); mockService.Setup(service => service.DoWork()).Returns(true);
```

---

23. **SOLID Principles**

Single Responsibility Principle (SRP)

- **Question:** What is the Single Responsibility Principle in C#?
  Answer: SRP states that a class should have only one reason to change, meaning it should have only one job or responsibility. This improves modularity and maintainability.

Dependency Inversion Principle (DIP)

- **Question:** How does the Dependency Inversion Principle (DIP) work in C#?
  Answer: DIP states that high-level modules should not depend on low-level modules; both should depend on abstractions. This is achieved through interfaces and dependency injection.

---

24. **Code Optimization and Performance Tuning**

Analyzing Memory Usage

- **Question:** How do you analyze memory usage in a C# application?
  Answer: Tools like Visual Studio's Diagnostic Tools and third-party profilers (e.g., dotMemory) help analyze memory usage by identifying memory leaks and high memory consumption.

---

25. **Architectural Patterns**

MVC (Model-View-Controller)

- **Question:** What is the MVC architectural pattern, and how is it implemented in C#?
  Answer: MVC (Model-View-Controller) separates an application into three components: Model (data), View (UI), and Controller (logic). It is commonly implemented in ASP.NET MVC and ASP.NET Core MVC applications.

Microservices Architecture

- **Question:** What is a microservices architecture, and how is it different from a monolithic architecture?
  Answer: Microservices architecture breaks down an application into loosely coupled, independently deployable services, each responsible for a specific business capability, unlike a monolithic architecture where all functionality is bundled together.

---

26. **Security in C#**

Encryption and Decryption

- **Question:** How is encryption and decryption implemented in C#?
  Answer: Encryption and decryption are implemented using the `System.Security.Cryptography` namespace, which provides algorithms like AES, RSA, and SHA for securing data.

  ```csharp
  using var aes = Aes.Create();
  ```

Secure Coding Practices

- **Question:** What are secure coding practices in C#?
  Answer: Secure coding practices include validating inputs, avoiding hardcoding sensitive information, using secure hashing algorithms, and ensuring proper exception handling to prevent security vulnerabilities.

---

27. **Version Control (Git and GitHub)**

Branching Strategies

- **Question:** What is Gitflow, and how does it work?
  Answer: Gitflow is a popular branching strategy that involves creating feature branches, release branches, and a main (production) branch, allowing for structured collaboration and release management.