**1.What are the six combinations of access modifier keywords and what do they do?**

C# supports six access modifiers, which determine the visibility of classes, methods, and members:

public – Accessible anywhere in the program.

private – Accessible only within the same class.

protected – Accessible in the same class and its derived (child) classes.

internal – Found in the same assembly (project).

protected internal – Visible within the same assembly or within derived classes in other assemblies. private protected – Accessible within the same class or inheriting classes of the same assembly (added in C# 7.2).

**2.What is the difference between the static, const, and readonly keywords when applied to  a type member?**

The static, const, and read-only keywords in C# have varying uses when used with type members:

**1.Static** – A class-level entity; it belongs to the **type** rather than any instance. Shared across all instances.

*Example:* A **static field** tracks the number of objects created.

```
public static int Object Count = 0;
```

**2.Const** – A **compile-time constant**; its value is set at declaration and **cannot change** afterward.

*Example:* Used for **mathematical constants** or **fixed values**.

public const double Pi = 3.14159;

**3.Readonly** – A **runtime constant**, meaning it can be assigned only **once** during initialization or in a constructor.

*Example:* Used for values that **depend on external factors** like **configuration settings**.

public readonly string ServerName;

public MyClass() { ServerName = "ProductionServer"; }

| Feature | Static | Const | **Readonly** |
|---|---|---|---|
| Scope | Belongs to **class** | Belongs to **class** | Belongs to **instance** |
| Mutability | Can **change** | **Immutable** (fixed at compile-time) | **Immutable** |
| When Assigned | Any time | At declaration only | In constructor or declaration |
| Example Usage | Global counters, utility methods | Fixed values like Pi, error codes | Configuration settings, runtime values |

3. What does a constructor do?

A constructor in C# is a special method which is automatically called when an object of the class is instantiated. It is utilized for initializing fields, assigning default values, and creating dependencies in an object. A constructor bears the same name as the class and does not return any type, not even void. Overloading of constructors is allowed in C#, i.e., more than one constructor can be present in a class with different parameters. Constructors come in different forms, such as default, parameterized, copy, and static constructors. A properly crafted constructor guarantees objects are instantiated with valid initial states, which makes code readable and maintainable

4. Why is the partial keyword useful?

The partial keyword in C# is convenient because it allows a class, struct, or interface to be split into multiple files for improved code organization and maintainability. This feature is particularly beneficial for large projects because it can permit different developers to work on different parts of the same class without interfering with each other. It is extensively used in auto-generated code, such as Windows Forms and Entity Framework, where tools might generate part of a class but would need to be adjusted manually. With partial, the compiler takes all parts as a single unit, ensuring that they fit together seamlessly without any hitch while having a clean and modular codebase.

5. What is a tuple?

A C# tuple is a small, immutable data structure that provides grouping of several values of varying types into one unit without the creation of a specific class or struct. It is most useful in returning more than one value from a method when the establishment of a new type would be redundant. Tuples are accessed by.Item1,.Item2,.Item3, and so on, but named tuples are more readable as elements are given meaningful names. Tuples are different from arrays as they can hold different types of data and retain value order. Tuples improve the efficiency of code, eliminate repetitive class definitions, and enhance readability in cases where temporary grouping of data is required.

6. What does the C# record keyword do?

Record in C# is immutable reference types that are intended for data storage and transfer, primarily. The two key differences of records from classes are: it has value-based equality: i.e., two records with the same values are equal instead of object references and records are immutable by default and the properties are only initialized at the time of declaration. Furthermore, they provide concise syntax which eliminates boilerplate code in declaring data models. Records are especially helpful in DTOs (Data Transfer Objects), data modeling, and functional programming, where immutability and predictability of data-oriented applications are guaranteed.

7. What does overloading and overriding mean?

Overriding and overloading are fundamental object-oriented programming concepts that allow method customization in C#. Overloading is where there are methods of the same name but different parameters under the same class, which allows compile-time polymorphism. This is used to have flexibility in the use of methods with varying inputs. On the other hand, overriding refers to the activity of a subclass altering the functionality of a superclass's method using the `override` keyword in order to achieve runtime polymorphism. Overriding ensures that a subclass provides a specific implementation of a method first defined in the superclass.

8. What is the difference between a field and a property?

A field and a property in C# both store data within a class but with different purposes. A field is a simple variable that stores data directly and typically is declared `private` to maintain encapsulation. Fields do not have additional logic and are mostly meant for storing internal data. Contrary to this, a property provides access to a field with control through ‘get’ and ‘set’ accessors where there is choice to validate, compute, or limit before the assignment or the retrieval of the values. Properties strengthen encapsulation and hence used preferably for regulating data access whereas fields are convenient for holding hidden data.

9. How do you make a method parameter optional?

In C#, an argument can be made optional by assigning a default value when the parameter is declared in the method. If no argument is passed for the parameter when the method is called, the default value is applied. This makes method calls easy because it provides flexibility when providing arguments. Optional arguments should be placed at the end of the list of parameters so that confusion does not arise. The second method for offering voluntary behavior is through method overloading, where multiple methods are defined with different lists of parameters. Named arguments can also be used for selective passing of values when calling the method.

10. What is an interface and how is it different from abstract class?

An interface in C# is more of a design used to define a set of methods, properties, or events that a class must implement itself and does not provide any implementation. It is mainly used to enforce a contract over multiple classes, achieving flexibility and multiple inheritance. On the other hand, an abstract class can have a mix of abstract methods, which are to be implemented by the derived classes, and concrete methods with predefined functionality. Abstract classes may include fields, constructors, and access modifiers too. Interfaces can't. An interface is all about the strict enforcement of behavior without implementation, while an abstract class facilitates code reuse and partial implementation.

11. What accessibility level are members of an interface?

In C#, all members of an interface are publically implied and cannot be specified with access modifiers. This means that methods, properties, and events defined in an interface must be implemented as public in the implementing class. Since interfaces should be used to represent a contract and not an implementation, members like private, protected, or internal are not allowed. In case an access modifier is stated explicitly, a compilation error is generated. The purpose of this design is to provide access to all members of the interface and enforce consistent implementation in more than one class.

12. True/False. Polymorphism allows derived classes to provide different implementations of the same method.

True

13. True/False. The override keyword is used to indicate that a method in a derived class is providing its own implementation of a method.

True

14. True/False. The new keyword is used to indicate that a method in a derived class is providing its own implementation of a method.

True

15. True/False. Abstract methods can be used in a normal (non-abstract) class.

False

16. True/False. Normal (non-abstract) methods can be used in an abstract class.

True

17. True/False. Derived classes can override methods that were virtual in the base class.

True.

18. True/False. Derived classes can override methods that were abstract in the base class.

True

19. True/False. In a derived class, you can override a method that was neither virtual non abstract in the base class.

False

20. True/False. A class that implements an interface does not have to provide an implementation for all of the members of the interface.

False

21. True/False. A class that implements an interface is allowed to have other members that aren’t defined in the interface.

True

22. True/False. A class can have more than one base class.

False

23. True/False. A class can implement more than one interface.

True