Assignment\_3\_OOP

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

* In C#, access modifiers control the visibility of classes, methods, fields, and other members. The six main modifiers are: public, private, protected, internal, protected internal, and private protected. Public members are accessible from anywhere, while private members are only accessible within their containing class. Protected members can be accessed within the class and by derived classes. Internal members are accessible only within the same assembly. Protected internal members combine protected and internal, making them accessible either within the same assembly or by derived classes in other assemblies. Finally, private protected members are accessible only by derived classes in the same assembly. These modifiers help enforce encapsulation and control how code components interact.

2.What is the difference between the static, const, and read-only keywords when applied to a type member?

* In C#, static, const, and readonly are keywords that control how type members are stored and assigned. A static member belongs to the class itself rather than any instance, meaning it is shared across all objects. A const member is a compile-time constant that must be assigned at declaration and cannot be changed; it is implicitly static. A readonly member can be assigned either at declaration or in a constructor, allowing its value to be set at runtime but preventing further modification. Unlike const, readonly can differ between instances unless it is also declared static. Together, these keywords help manage data immutability, sharing, and encapsulation in C# programs.

3. What does a constructor do?

* In C#, a constructor is a special method that is automatically called when an instance of a class is created. Its main purpose is to initialize the object’s fields or properties and set up any necessary state before the object is used. Constructors have the same name as the class, do not have a return type, and can be overloaded to provide different ways of initializing objects. They can also call other constructors within the same class or base class to reuse initialization logic. Overall, constructors ensure that objects start in a valid and consistent state.

4. Why is the partial keyword useful?

* In C#, the partial keyword allows a class, struct, or interface to be split across multiple files, while still being treated as a single type by the compiler. This is useful for organizing large classes, separating auto-generated code from developer-written code, and enabling multiple developers to work on the same type simultaneously without conflicts. By using partial, developers can improve code maintainability, readability, and collaboration in large projects.

5. What is a tuple?

* In C#, a tuple is a lightweight, ordered collection of multiple values that can be of different types. Tuples allow you to group related data together without creating a separate class or struct. They can be named or unnamed, and their elements are accessed using Item1, Item2, etc., or by custom names.

6. What does the C# record keyword do?

* In C#, the record keyword is used to define a reference type with built-in support for immutability, value-based equality, and concise syntax. Records are ideal for data-centric classes where the main purpose is to store data rather than behavior.

7. What does overloading and overriding mean?

* In C#, overloading allows a class to have multiple methods with the same name but different parameters, enabling the compiler to choose the correct method at compile time. Overriding allows a derived class to provide a new implementation of a base class method marked as virtual or abstract, enabling the runtime to determine which method to execute. While overloading supports compile-time polymorphism, overriding supports runtime polymorphism, allowing derived classes to customize or extend base class behavior.

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

* In C#, a field is a variable that is declared directly in a class or struct to store data, whereas a property is a member that provides controlled access to a field through getters and setters. Fields are usually private to enforce encapsulation, while properties can define logic when getting or setting values, such as validation or computed values. Properties also allow for data binding in frameworks like WPF or ASP.NET. In short, fields hold data, and properties provide controlled, often public, access to that data.

9. How do you make a method parameter optional?

* In C#, you can make a method parameter optional by providing a default value in the method declaration. If the caller does not supply an argument for that parameter, the default value is used.

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

* An interface in C# defines a contract of methods, properties, and events that implementing classes must fulfill, without providing implementation, while an abstract class can provide both abstract members and concrete implementations. Interfaces allow multiple inheritance and are ideal for defining capabilities across unrelated classes, whereas abstract classes provide a shared base with common functionality for related classes.

11. What accessibility level are members of an interface?

* In C#, all members of an interface are implicitly public. You cannot specify an access modifier (like private, protected, or internal) for interface members; they are always meant to be implemented and accessible by any class that implements the interface.

12. True/False. Polymorphism allows derived classes to provide different implementations

of the same method.

* True, polymorphism in C# allows derived classes to provide different implementations of a method defined in a base class, typically using method overriding with virtual and override keywords. This enables runtime behavior to depend on the actual object type, even when accessed through a base class reference.

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, the override keyword is used in a derived class to provide a new implementation of a method that is marked virtual or abstract in the base class. This enables runtime polymorphism, allowing the derived class’s version of the method to be called when accessed through a base class reference.

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, the new keyword can be used in a derived class to hide a method in the base class by providing a new implementation. Unlike override, this does not participate in runtime polymorphism; it simply hides the base class method when accessed through the derived class.

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

* False, abstract methods can only be declared in abstract classes. A normal (non-abstract) class cannot contain abstract methods because abstract methods do not have an implementation, and non-abstract classes must provide complete implementations for all their members.

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

* True, an abstract class can contain both abstract methods (without implementation) and normal (non-abstract) methods with full implementations. Normal methods in an abstract class can provide common functionality that derived classes can use or override.

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

* True, derived classes can override methods that are marked virtual in the base class by using the override keyword. This allows the derived class to provide a custom implementation while enabling runtime polymorphism.

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

* True, derived classes must override abstract methods defined in a base class. Abstract methods have no implementation in the base class, so the derived class is required to provide a concrete implementation.

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

* False, you cannot override a method in a derived class if it is neither virtual nor abstract in the base class. Only methods marked as virtual, abstract, or already override can be overridden. Non-virtual methods can be hidden using the new keyword, but this is not overriding.

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, a class that implements an interface must provide an implementation for all members of the interface. If it does not, the class must be declared abstract. This ensures that the class fulfills the contract defined by the interface.

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

* True, a class that implements an interface can have additional members (fields, properties, methods) that are not defined in the interface. Implementing an interface only requires that the class provides implementations for the interface members, but it can include any other members needed for its functionality.

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

* False, a class cannot inherit from more than one base class because C# does not support multiple class inheritance. A class can, however, implement multiple interfaces, which allows it to inherit behavior from multiple sources indirectly.

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

* True, a class can implement multiple interfaces, allowing it to inherit the contracts of more than one interface. This enables a class to provide implementations for various sets of methods or properties defined in different interfaces, promoting flexibility and modular design.