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

* **Public**: Allows access from any other code in the same assembly or another assembly.
* **Private**: Restricts access to only within the containing type.
* **Protected**: Allows access within the containing type and any derived types.
* **Internal**: Provides access within the same assembly, but not to other assemblies.
* **Protected Internal**: Accessible within the same assembly or from derived types in other assemblies.
* **Private Protected**: Accessible within the same assembly or from derived types in the same assembly, but not from other assemblies.

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

* **Static**: Indicates that the member belongs to the type itself rather than an instance of the type.
* **Const**: Defines a constant value that cannot be changed.
* **Readonly**: Allows a value to be assigned once and not changed thereafter, typically set at construction or declaration.

1. What does a constructor do?

**Constructor**: A constructor is a special method that is automatically called when an instance of a class is created. It is used to initialize the object's state.

1. Why is the partial keyword useful?

The **partial** keyword allows a class, struct, interface, or method to be defined across multiple files. It's useful for separating large classes into manageable parts or for code generation scenarios.

1. What is a tuple?

A tuple is a data structure in C# that can hold multiple elements of potentially different types in a single object without needing to create a separate class to hold them.

1. What does the C# record keyword do?

Introduced in C# 9.0, the **record** keyword defines a reference type that provides built-in functionality for immutable data modeling.

1. What does overloading and overriding mean?

**Overloading**: Refers to defining multiple methods with the same name but different parameters within the same class. They must differ in the type or number of their parameters.

**Overriding**: Involves providing a new implementation for a base class method in a derived class using the **override** keyword. It allows polymorphic behavior based on the object instance.

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

**Field**: Represents a variable that is part of a class or struct and stores data.

**Property**: Provides a flexible mechanism to read, write, or compute the value of a private field. It exposes access to the field while controlling how it is accessed and modified.

1. How do you make a method parameter optional?

You can make a method parameter optional by providing a default value for it in the method signature. For example, **void Method(int x = 10)** makes **x** optional with a default value of 10.

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

**Interface**: Defines a contract for classes to implement without providing any implementation details. It can only contain method signatures, properties, events, and indexers.

**Abstract Class**: Can contain both abstract (unimplemented) and concrete (implemented) members. It allows defining some default behavior and can have fields, constructors, and events.

1. What accessibility level are members of an interface?

Interface members are implicitly **public** and cannot have any access modifiers specified.

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.

False. The new keyword in C# is used to explicitly hide a member of the base class in the derived class, not to provide its own implementation of a method from the base class.

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

False. Abstract methods can only exist in abstract classes or interfaces. They define a method signature that must be implemented by any non-abstract derived class.

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. Derived classes **must** override methods that were declared as **abstract** in the base class. Abstract methods are intended to be implemented by derived classes to provide specific functionality.

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

False. Overriding requires the method in the base class to be explicitly marked as virtual or abstract.

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

True.

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.

Coding questions:

* 1. Reverse array:

using System;

class Program

{

static void Main(string[] args)

{

int[] numbers = GenerateNumbers(10); // Generate an array of 10 numbers

Console.WriteLine("Original array:");

PrintNumbers(numbers); // Print original array

Reverse(numbers); // Reverse the array

Console.WriteLine("\nReversed array:");

PrintNumbers(numbers); // Print reversed array

}

static int[] GenerateNumbers(int length)

{

int[] numbers = new int[length];

for (int i = 0; i < length; i++)

{

numbers[i] = i + 1; // Fill the array with numbers 1 to length

}

return numbers;

}

static void PrintNumbers(int[] array)

{

foreach (int num in array)

{

Console.Write(num + " ");

}

Console.WriteLine();

}

static void Reverse(int[] array)

{

// Hint #2: Use a loop to swap elements from the start and end of the array

for (int i = 0; i < array.Length / 2; i++)

{

int temp = array[i];

array[i] = array[array.Length - i - 1];

array[array.Length - i - 1] = temp;

}

}

}

* 1. Fibonacci numbers

using System;

class Program

{

static void Main(string[] args)

{

// Loop through the first 10 numbers of the Fibonacci sequence and print them

for (int i = 1; i <= 10; i++)

{

int fibNumber = Fibonacci(i);

Console.WriteLine($"Fibonacci({i}) = {fibNumber}");

}

}

static int Fibonacci(int n)

{

// Base cases

if (n == 1 || n == 2)

{

return 1;

}

// Recursive case

return Fibonacci(n - 1) + Fibonacci(n - 2);

}

}

Design and build class:

* 1. Four principles of oop

using System;

// Abstraction: Abstract class Animal

public abstract class Animal

{

public abstract void MakeSound(); // Abstract method representing making a sound

}

// Encapsulation: Concrete class Dog encapsulating its name

public class Dog : Animal

{

private string name;

// Constructor to initialize Dog with a name

public Dog(string name)

{

this.name = name;

}

// Polymorphism: Implementing MakeSound() method for Dog

public override void MakeSound()

{

Console.WriteLine($"{name} says Woof!");

}

}

// Inheritance: Concrete class Cat inheriting from Animal

public class Cat : Animal

{

// Polymorphism: Implementing MakeSound() method for Cat

public override void MakeSound()

{

Console.WriteLine("Meow!");

}

}

class Program

{

static void Main(string[] args)

{

// Creating instances of Dog and Cat

Dog dog = new Dog("Buddy");

Cat cat = new Cat();

// Demonstrating Polymorphism by calling MakeSound() on different objects

dog.MakeSound();

cat.MakeSound();

}

}

* 1. Abstraction for student and instructor:

using System;

// Abstraction: Abstract class Person

public abstract class Person

{

public string Name { get; set; } // Property to store the name of the person

// Abstract method to define behavior specific to each person type

public abstract void Introduce();

}

// Student class inheriting from Person

public class Student : Person

{

public int GradeLevel { get; set; } // Property to store the grade level of the student

// Constructor to initialize Student with a name and grade level

public Student(string name, int gradeLevel)

{

Name = name;

GradeLevel = gradeLevel;

}

// Implementation of Introduce() method for Student

public override void Introduce()

{

Console.WriteLine($"Hi, I'm {Name}, a student in grade {GradeLevel}.");

}

// Additional behavior specific to Student can be added here

}

// Instructor class inheriting from Person

public class Instructor : Person

{

public string Subject { get; set; } // Property to store the subject taught by the instructor

// Constructor to initialize Instructor with a name and subject

public Instructor(string name, string subject)

{

Name = name;

Subject = subject;

}

// Implementation of Introduce() method for Instructor

public override void Introduce()

{

Console.WriteLine($"Hello, I'm {Name}, an instructor teaching {Subject}.");

}

// Additional behavior specific to Instructor can be added here

}

class Program

{

static void Main(string[] args)

{

// Creating instances of Student and Instructor

Student student = new Student("Alice", 8);

Instructor instructor = new Instructor("Mr. Smith", "Math");

// Using polymorphism to call Introduce() method on different objects

student.Introduce();

instructor.Introduce();

}

}

* 1. Encapsulation

using System;

// Example: Person class demonstrating Encapsulation

public class Person

{

// Private fields encapsulated to ensure access through properties

private string name;

private int age;

// Constructor to initialize Person details

public Person(string name, int age)

{

this.name = name;

this.age = age;

}

// Public property to access the person's name

public string Name

{

get { return name; }

set { name = value; } // Setter allows modification of the name

}

// Public property to access the person's age

public int Age

{

get { return age; }

set { age = value; } // Setter allows modification of the age

}

// Public method to display details of the person

public void DisplayDetails()

{

Console.WriteLine($"Person: {Name}, Age: {Age}");

}

}

// Example: Account class demonstrating Encapsulation

public class Account

{

// Private fields encapsulated to ensure access through properties

private string accountNumber;

private decimal balance;

// Constructor to initialize Account details

public Account(string accountNumber, decimal balance)

{

this.accountNumber = accountNumber;

this.balance = balance;

}

// Public property to access the account number

public string AccountNumber

{

get { return accountNumber; }

}

// Public property to access the account balance

public decimal Balance

{

get { return balance; }

set { balance = value; } // Setter allows modification of the balance

}

// Public method to display account information

public void DisplayAccountInfo()

{

Console.WriteLine($"Account Number: {AccountNumber}, Balance: {Balance:C}");

}

}

class Program

{

static void Main(string[] args)

{

// Example 1: Using Person class

Person person = new Person("Alice", 30);

person.DisplayDetails();

// Modifying person's age through property (Encapsulation)

person.Age = 32;

Console.WriteLine($"Updated Age: {person.Age}");

Console.WriteLine();

// Example 2: Using Account class

Account account = new Account("123456789", 1000.50m);

account.DisplayAccountInfo();

// Modifying account balance through property (Encapsulation)

account.Balance += 500;

Console.WriteLine($"Updated Balance: {account.Balance:C}");

}

}

* 1. Inheritance

using System;

// Base class: Person demonstrating Inheritance

public class Person

{

// Protected fields accessible to derived classes

protected string name;

protected int age;

// Constructor to initialize Person details

public Person(string name, int age)

{

this.name = name;

this.age = age;

}

// Public method to display details of the person

public virtual void DisplayDetails()

{

Console.WriteLine($"Person: {name}, Age: {age}");

}

}

// Derived class: Student inheriting from Person

public class Student : Person

{

public int GradeLevel { get; set; } // Property specific to Student

// Constructor to initialize Student details

public Student(string name, int age, int gradeLevel) : base(name, age)

{

GradeLevel = gradeLevel;

}

// Override method to display details of the student

public override void DisplayDetails()

{

base.DisplayDetails();

Console.WriteLine($"Grade Level: {GradeLevel}");

}

}

// Derived class: Instructor inheriting from Person

public class Instructor : Person

{

public string Subject { get; set; } // Property specific to Instructor

// Constructor to initialize Instructor details

public Instructor(string name, int age, string subject) : base(name, age)

{

Subject = subject;

}

// Override method to display details of the instructor

public override void DisplayDetails()

{

base.DisplayDetails();

Console.WriteLine($"Teaches: {Subject}");

}

}

class Program

{

static void Main(string[] args)

{

// Using Student class

Student student = new Student("Alice", 20, 10);

student.DisplayDetails();

Console.WriteLine();

// Using Instructor class

Instructor instructor = new Instructor("Mr. Smith", 35, "Math");

instructor.DisplayDetails();

}

}

* 1. Polymorphism

using System;

// Base class: Person demonstrating Inheritance and Polymorphism

public class Person

{

protected string name;

protected int age;

// Constructor to initialize Person details

public Person(string name, int age)

{

this.name = name;

this.age = age;

}

// Virtual method for displaying details, can be overridden

public virtual void DisplayDetails()

{

Console.WriteLine($"Person: {name}, Age: {age}");

}

// Virtual method for calculating salary, to be overridden

public virtual decimal CalculateSalary()

{

return 0; // Default implementation, to be overridden

}

}

// Derived class: Student inheriting from Person

public class Student : Person

{

public int GradeLevel { get; set; } // Property specific to Student

// Constructor to initialize Student details

public Student(string name, int age, int gradeLevel) : base(name, age)

{

GradeLevel = gradeLevel;

}

// Override method to display details of the student

public override void DisplayDetails()

{

base.DisplayDetails();

Console.WriteLine($"Grade Level: {GradeLevel}");

}

}

// Derived class: Instructor inheriting from Person

public class Instructor : Person

{

public string Subject { get; set; } // Property specific to Instructor

public decimal Salary { get; set; } // Property specific to Instructor

// Constructor to initialize Instructor details

public Instructor(string name, int age, string subject, decimal salary) : base(name, age)

{

Subject = subject;

Salary = salary;

}

// Override method to display details of the instructor

public override void DisplayDetails()

{

base.DisplayDetails();

Console.WriteLine($"Teaches: {Subject}");

}

// Override method to calculate salary for Instructor

public override decimal CalculateSalary()

{

// Example: Salary calculation based on subject taught

decimal baseSalary = 3000; // Base salary

decimal subjectBonus = 500; // Bonus for teaching a specific subject

// Example: Specific calculation based on subject taught

if (Subject == "Math")

{

return baseSalary + subjectBonus;

}

else

{

return baseSalary;

}

}

}

class Program

{

static void Main(string[] args)

{

// Using Student class

Student student = new Student("Alice", 20, 10);

student.DisplayDetails();

Console.WriteLine();

// Using Instructor class

Instructor instructor = new Instructor("Mr. Smith", 35, "Math", 0);

instructor.DisplayDetails();

// Calculating and displaying salary for Instructor using polymorphism

decimal salary = instructor.CalculateSalary();

Console.WriteLine($"Salary: {salary:C}");

}

}

* 1. Interface:

using System;

using System.Collections.Generic;

// Interface for basic person operations

public interface IPersonService

{

int CalculateAge(DateTime birthDate);

decimal CalculateSalary(decimal baseSalary);

List<string> GetAddresses();

}

// Interface for course-related operations

public interface ICourseService

{

List<Student> EnrolledStudents { get; }

}

// Interface for student-specific operations

public interface IStudentService : IPersonService

{

decimal CalculateGPA(Dictionary<string, string> grades);

}

// Interface for instructor-specific operations

public interface IInstructorService : IPersonService

{

string Department { get; set; }

DateTime JoinDate { get; set; }

decimal CalculateBonusSalary();

int CalculateYearsOfExperience();

}

// Interface for department-related operations

public interface IDepartmentService

{

Instructor DepartmentHead { get; set; }

decimal Budget { get; set; }

List<Course> OfferedCourses { get; set; }

}

* 1. Two classes

public class Color

{

private int red;

private int green;

private int blue;

private int alpha;

// Constructor with red, green, blue, and alpha values

public Color(int red, int green, int blue, int alpha)

{

this.red = red;

this.green = green;

this.blue = blue;

this.alpha = alpha;

}

// Constructor with red, green, and blue values (alpha defaults to 255)

public Color(int red, int green, int blue) : this(red, green, blue, 255)

{

}

// Methods to get and set the color components

public int Red

{

get { return red; }

set { red = value; }

}

public int Green

{

get { return green; }

set { green = value; }

}

public int Blue

{

get { return blue; }

set { blue = value; }

}

public int Alpha

{

get { return alpha; }

set { alpha = value; }

}

// Method to calculate grayscale value

public double GetGrayscale()

{

return (red + green + blue) / 3.0;

}

}

public class Ball

{

private int size;

private Color color;

private int throwCount;

private bool popped;

// Constructor with size and color

public Ball(int size, Color color)

{

this.size = size;

this.color = color;

throwCount = 0;

popped = false;

}

// Constructor with size and color (color defaults to black with full opacity)

public Ball(int size) : this(size, new Color(0, 0, 0, 255))

{

}

// Pop method sets size to 0

public void Pop()

{

size = 0;

popped = true;

}

// Throw method increments throw count if ball is not popped

public void Throw()

{

if (!popped)

{

throwCount++;

}

}

// Method to get the number of times the ball has been thrown

public int GetThrowCount()

{

return throwCount;

}

}

class Program

{

static void Main(string[] args)

{

// Create some balls with different colors

Color redColor = new Color(255, 0, 0);

Color blueColor = new Color(0, 0, 255);

Ball redBall = new Ball(10, redColor);

Ball blueBall = new Ball(15, blueColor);

// Throw the balls a few times

redBall.Throw();

redBall.Throw();

blueBall.Throw();

blueBall.Throw();

blueBall.Throw();

// Pop one ball

redBall.Pop();

// Try to throw the balls again

redBall.Throw();

blueBall.Throw();

// Display the throw counts

Console.WriteLine($"Red Ball throws: {redBall.GetThrowCount()}");

Console.WriteLine($"Blue Ball throws: {blueBall.GetThrowCount()}");

}

}