"*Working With Methods: Questions 1 and 2 is not required.*
"Designing and Building Classes using object-oriented principles" Question 7 is not required.

Test your knowledge

1. What are the six combinations of access modifier keywords and what do they do?
By default, if no access modifier is specified, the member is private.

- public: Access is not restricted, and members can be accessed from anywhere in the code.
- private: Access is limited to within the class in which it is declared. Members cannot be accessed outside the class.
- protected: Access is limited to within the class in which it is declared and any subclass of that class.
- internal: Access is limited to within the same assembly (compilation unit).
- protected internal: Access is limited to within the same assembly and to derived classes within any assembly.
- private protected: Access is limited to within the class in which it is declared, and to derived classes within the same assembly.

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

- const and readonly fields are similar in that they both prevent the value of the field from being changed after it is assigned
- but const fields are evaluated at compile time, while readonly fields can be assigned at runtime.
- The static keyword, on the other hand, changes the way the field or property is shared among instances of the class.

3. What does a constructor do?
A constructor in C# is a special method that is automatically called when an instance of a class is created. It is used to initialize the state of the object, and to allocate any necessary resources. The constructor has the same name as the class, and has no return type (not even void).

4. Why is the partial keyword useful?
The partial keyword is useful in C# because it enables you to split a class, structure, or interface definition into multiple files, making it easier to manage and maintain large class definitions, to work with code generators, and to distribute the development of a single class across multiple developers.

5. What is a tuple?
Tuple is a lightweight data structure that enables you to store multiple values of different data types in a single object. A tuple is similar to a class or a struct, but is designed to be more concise and easier to use.

6. What does the C# record keyword do?
It provides a way to define value-based classes. A value-based class is a type that represents a value, rather than an identity.

7. What does overloading and overriding mean?
- Function overloading is a regular function, but it is assigned with multiple parameters. It allows the creation of several methods with the same name which differ from each other by the type of input and output of the function.
- Method overriding is a feature that allows a subclass to provide the implementation of a method that overrides in the main class. It will override the implementation in the superclass by providing the same method name, same parameter, and same return type.

8. What is the difference between a field and a property?
The main difference between a field and a property in C# is that fields are directly accessible variables, whereas properties are methods that encapsulate data and control how it is accessed and modified.

9. How do you make a method parameter optional?
We can make a method parameter optional by providing a default value for the parameter in the method signature. When the method is called, if a value is not supplied for the optional parameter, the default value will be used instead.

10. What is an interface and how is it different from abstract class?
An interface defines a set of methods that a class must implement, which contains methods, but not their definition. It's a collection of the abstract method.

An interface is similar to a class, but it does not contain any implementation code for its methods. Abstract classes, on the other hand, can provide an implementation of their members, which can be inherited by derived classes.

11. What accessibility level are members of an interface?

Members of an interface are always public. This means that any class that implements an interface must provide a public implementation for all of the members defined in the interface.

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

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.

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.

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

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

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

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

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

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

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

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

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

Designing and Building Classes using object-oriented principles

1. Write a program that that demonstrates use of four basic principles of object-oriented programming /Abstraction/, /Encapsulation/, /Inheritance/ and

/Polymorphism/.

```
using System;

// Abstraction
abstract class Animal
{
    // Encapsulation
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    // Abstraction
    public abstract void MakeSound();
}

// Inheritance
class Dog : Animal
{
    // Polymorphism
    public override void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}

class Cat : Animal
{
    // Polymorphism
    public override void MakeSound()
    {
        Console.WriteLine("Meow!");
    }
}

class Program
{
    static void Main(string[] args)
```

```
    {
        Dog dog = new Dog();
        dog.Name = "Duke";
        Console.WriteLine("Dog's Name: " + dog.Name);
        dog.MakeSound();

        Cat cat = new Cat();
        cat.Name = "Lily";
        Console.WriteLine("Cat's Name: " + cat.Name);
        cat.MakeSound();

        Console.ReadLine();
    }
}
```

2. Use /Abstraction/ to define different classes for each person type such as Student and Instructor. These classes should have behavior for that type of person.

```
abstract class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Address { get; set; }

    public abstract void Work();
}

class Student: Person
{
    public int StudentID { get; set; }
    public string Course { get; set; }

    public override void Work()
    {
        Console.WriteLine("Studying");
    }
}

class Instructor: Person
{
    public int EmployeeID { get; set; }
```

```
    public string Department { get; set; }

    public override void Work()
    {
        Console.WriteLine("Teaching");
    }
}
```

3. Use /Encapsulation/ to keep many details private in each class.

```
class Person
{
    private string name;
    private int age;
    private string address;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int Age
    {
        get { return age; }
        set { age = value; }
    }

    public string Address
    {
        get { return address; }
        set { address = value; }
    }

    public virtual void Work()
    {
        Console.WriteLine("Working");
    }
}

class Student: Person
{
```

```
    private int studentID;
    private string course;

    public int StudentID
    {
        get { return studentID; }
        set { studentId = value; }
    }

    public string Course
    {
        get { return course; }
        set { course = value; }
    }

    public override void Work()
    {
        Console.WriteLine("Studying");
    }
}

class Instructor: Person
{
    private int employeeID;
    private string department;

    public int EmployeeID
    {
        get { return employeeID; }
        set { employeeID = value; }
    }

    public string Department
    {
        get { return department; }
        set { department = value; }
    }

    public override void Work()
    {
        Console.WriteLine("Teaching");
    }
```

```
}
```

4. Use /Inheritance/ by leveraging the implementation already created in the Person class to save code in Student and Instructor classes.

```
public class Person
{
    private string name;
    private int age;

    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int Age
    {
        get { return age; }
        set { age = value; }
    }

    public virtual void PrintDetails()
    {
        Console.WriteLine("Name: " + name + ", Age: " +
age);
    }
}

public class Student: Person
{
    private string major;

    public Student(string name, int age, string major):
base(name, age)
    {
```

```
        this.major = major;
    }

    public string Major
    {
        get { return major; }
        set { major = value; }
    }

    public override void PrintDetails()
    {
        base.PrintDetails();
        Console.WriteLine("Major: " + major);
    }
}

public class Instructor: Person
{
    private string subject;

    public Instructor(string name, int age, string
subject): base(name, age)
    {
        this.subject = subject;
    }

    public string Subject
    {
        get { return subject; }
        set { subject = value; }
    }

    public override void PrintDetails()
    {
        base.PrintDetails();
        Console.WriteLine("Subject: " + subject);
    }
}
```

5. Use /Polymorphism/ to create virtual methods that derived classes could override to create specific behavior such as salary calculations.

```
using System;

// Define the base class, Person, with common behavior and
properties
public abstract class Person
{
    private string name;
    private int age;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int Age
    {
        get { return age; }
        set { age = value; }
    }

    // Virtual method that derived classes can override
    public virtual int CalculateSalary()
    {
        return 0;
    }
}

// Define the derived class, Student, with specific behavior
public class Student: Person
{
    private int studentId;

    public int StudentId
    {
        get { return studentId; }
        set { studentId = value; }
    }

    // Overriding the virtual method to provide specific
behavior
    public override int CalculateSalary()
    {
        return 0;
    }
}
```

```
// Define the derived class, Instructor, with specific behavior
public class Instructor: Person
{
    private int instructorId;
    private int hourlyRate;

    public int InstructorId
    {
        get { return instructorId; }
        set { instructorId = value; }
    }

    public int HourlyRate
    {
        get { return hourlyRate; }
        set { hourlyRate = value; }
    }

    // Overriding the virtual method to provide specific
behavior
    public override int CalculateSalary()
    {
        return HourlyRate * 8 * 42;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Student student = new Student();
        student.Name = "Jack Ma";
        student.Age = 25;
        student.StudentId = 54321;

        Console.WriteLine("Student name: " + student.Name);
        Console.WriteLine("Student age: " + student.Age);
        Console.WriteLine("Student ID: " + student.StudentId);
        Console.WriteLine("Student salary: " +
student.CalculateSalary());

        Console.WriteLine();

        Instructor instructor = new Instructor();
        instructor.Name = "John Doe";
        instructor.Age = 35;
        instructor.InstructorId = 12345;
```

```
        instructor.HourlyRate = 35;

        Console.WriteLine("Instructor name: " +
instructor.Name);
        Console.WriteLine("Instructor age: " + instructor.Age);
        Console.WriteLine("Instructor ID: " +
instructor.InstructorId);
        Console.WriteLine("Instructor salary: " +
instructor.CalculateSalary());
    }
}
```

6. Make sure to create appropriate /interfaces/ such as ICourseService, IStudentService, IInstructorService, IDepartmentService, IPersonService, IPersonService (should have person specific methods). IStudentService, IInstructorService should inherit from IPersonService.

Person

Calculate Age of the Person

Calculate the Salary of the person, Use decimal for salary

Salary cannot be negative number

Can have multiple Addresses, should have method to get addresses

Instructor

Belongs to one Department and he can be Head of the Department

Instructor will have added bonus salary based on his experience, calculate his years of experience based on Join Date

Student

Can take multiple courses

Calculate student GPA based on grades for courses

Each course will have grade from A to F

Course

Will have list of enrolled students

Department

Will have one Instructor as head

Will have Budget for school year (start and end Date Time)

Will offer list of courses

```
using System;
using System.Collections.Generic;

namespace Assignment3
{
    public interface IPersonService
    {
        int CalculateAge();
        decimal CalculateSalary();
        List<string> GetAddresses();
    }

    public interface IInstructorService: IPersonService
    {
    string Department { get; set; }
    bool IsHeadOfDepartment { get; set; }
    decimal BonusSalary { get; set; }
    int YearsOfExperience { get; }
    }

    interface IStudentService: IPersonService
    {
    List<(string courseName, char grade)> Courses { get; set; }
    decimal CalculateGPA();
    }

    public interface ICourseService
    {
    List<Student> EnrolledStudents { get; set; }
    }

    interface IDepartmentService
    {
    IInstructor Head { get; set; }
    decimal Budget { get; set; }
    DateTime StartDate { get; set; }
    DateTime EndDate { get; set; }
```

```
    List<ICourse> Courses { get; set; }
    }
}



using System;
using System.Collections.Generic;

public class Person
{
    // Abstraction - classes are defined to show specific
behavior for that type of person
    private DateTime dateOfBirth;
    private List<string> addresses = new List<string>();
    private decimal salary;

    // Encapsulation - some details are kept private
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public decimal Salary
    {
        get { return salary; }
        set
        {
            if (value < 0)
            {
                throw new ArgumentException("Salary cannot be
negative.");
            }
            salary = value;
        }
    }

    public DateTime DateOfBirth
    {
        get { return dateOfBirth; }
        set { dateOfBirth = value; }
    }

    public List<string> Addresses
    {
        get { return addresses; }
        set { addresses = value; }
    }
```

```
    public int CalculateAge()
    {
        int age = DateTime.Now.Year - dateOfBirth.Year;
        if (DateTime.Now.DayOfYear < dateOfBirth.DayOfYear)
        {
            age = age - 1;
        }
        return age;
    }

    // Polymorphism - virtual method can be overridden by
derived classes
    public virtual decimal CalculateSalary()
    {
        return salary;
    }

    public string GetAddresses()
    {
        string addressesString = "";
        foreach (string address in addresses)
        {
            addressesString += address + "\n";
        }
        return addressesString;
    }
}

public class Student: Person
{
    public string SchoolName { get; set; }

    public override decimal CalculateSalary()
    {
        return 0;
    }
}

public class Instructor: Person
{
    public string Department { get; set; }

    public override decimal CalculateSalary()
    {
        return salary;
    }
}
```