Lean Andrew B. Santos

BSIT 32E2 (IRREG)

**Part 1: C# (30 points)**

(10 points) Write a C# program that calculates the area of a triangle given its base and height. Include user input for both values and display the calculated area.

using System;

class Program

{

static void Main()

{

// Get user input for base and height

Console.Write("Enter the base of the triangle: ");

double baseLength = Convert.ToDouble(Console.ReadLine());

Console.Write("Enter the height of the triangle: ");

double height = Convert.ToDouble(Console.ReadLine());

// Calculate the area of the triangle

double area = CalculateTriangleArea(baseLength, height);

// Display the calculated area

Console.WriteLine($"The area of the triangle with base {baseLength} and height {height} is: {area}");

}

// Function to calculate the area of a triangle

static double CalculateTriangleArea(double baseLength, double height)

{

return 0.5 \* baseLength \* height;

}

}

**(10 points) Declare an array of 5 integers and fill it with values based on a user-defined formula (e.g., n^2). Then, print the largest element in the array.**

using System;

class Program

{

static void Main()

{

// Declare an array of 5 integers

int[] numbers = new int[5];

// Fill the array with values based on the user-defined formula (n^2)

Console.WriteLine("Filling the array based on the formula n^2:");

for (int i = 0; i < numbers.Length; i++)

{

numbers[i] = (i + 1) \* (i + 1);

Console.WriteLine($"Element {i + 1}: {numbers[i]}");

}

// Find and print the largest element in the array

int largestElement = FindLargestElement(numbers);

Console.WriteLine($"\nThe largest element in the array is: {largestElement}");

}

// Function to find the largest element in an array

static int FindLargestElement(int[] arr)

{

int max = arr[0];

for (int i = 1; i < arr.Length; i++)

{

if (arr[i] > max)

{

max = arr[i];

}

}

return max;

}

}

**(10 points) Implement a simple for loop that iterates from 1 to 10 and prints each number along with its square root.**

using System;

class Program

{

static void Main()

{

Console.WriteLine("Number\tSquare Root");

Console.WriteLine("-------------------");

// Simple for loop to iterate from 1 to 10

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

{

double squareRoot = Math.Sqrt(i);

Console.WriteLine($"{i}\t{squareRoot}");

}

}

}

**Part 2: HTML, CSS, and JavaScript (30 points)**

**HTML (10 points):** You are provided with the following incomplete HTML code snippet:

**HTML**

**<!DOCTYPE html>**

**<html>**

**<head>**

**<title>My Website</title>**

**</head>**

**<body>**

**<h1>Welcome to...</h1>**

**<p>This is a paragraph...</p>**

**<ul>**

**<li>Item 1</li>**

**<li>Item 2</li>**

**</ul>**

**</body>**

**</html>**

Complete the code snippet by adding the following elements:

An image within the <body> tag with a relevant src attribute.

An ordered list (<ol>) with three items.

A hyperlink within a <p> tag that points to an external website.

A CSS styling rule using an inline style attribute to change the font color of the <h3> heading.

CSS (10 points): Create a CSS stylesheet that defines the following styles:

Change the background color of the body element to light blue.

Apply a padding of 20px to all headings (h1, h2, h3).

Set the font size of the <p> tag to 14px.

Make the list items (li) have a bullet point style instead of the default numbers.

<!DOCTYPE html>

<html>

<head>

<title>My Website</title>

<style>

body {

background-color: lightblue;

}

h1, h2, h3 {

padding: 20px;

}

p {

font-size: 14px;

}

li {

list-style-type: disc;

}

h3 {

color: green;

}

</style>

</head>

<body>

<h1>Welcome to...</h1>

<p>This is a paragraph...</p>

<ul>

<li>Item 1</li>

<li>Item 2</li>

</ul>

<!-- An image within the <body> tag with a relevant src attribute.-->

<img src="Image.jpg" alt="Image Description">

<!-- An ordered list (<ol>) with three items-->

<ol>

<li>Item A</li>

<li>Item B</li>

<li>Item C</li>

</ol>

<!-- A hyperlink within a <p> tag that points to an external website.-->

<p>Visit our website: <a href="https://www.example.com" target="\_blank">Example Website</a></p>

<!-- <h3> heading with inline style to change font color -->

<h3 style="color: blue;">Your Subheading</h3>

</body>

</html>

**JavaScript (10 points):** Write a JavaScript function that takes a number as input and returns a string indicating whether the number is even or odd. Then, add a button to your HTML page that, when clicked, calls this function and displays the result (even or odd) in a paragraph element below the button.

<!DOCTYPE html>

<html>

<head>

<title>Even or Odd Checker</title>

<script>

// JavaScript function to check if a number is even or odd

function checkEvenOrOdd() {

// Get the input value from the user

var inputValue = document.getElementById("numberInput").value;

// Check if the input is a valid number

if (!isNaN(inputValue)) {

// Convert the input to a number

var number = parseInt(inputValue);

// Check if the number is even or odd

var result = (number % 2 === 0) ? "Even" : "Odd";

// Display the result in a paragraph element

document.getElementById("resultParagraph").innerHTML = "The number is " + result + ".";

} else {

// Display an error message if the input is not a valid number

document.getElementById("resultParagraph").innerHTML = "Please enter a valid number.";

}

}

</script>

</head>

<body>

<h1>Even or Odd Checker</h1>

<!-- Input field for the user to enter a number -->

<label for="numberInput">Enter a number:</label>

<input type="text" id="numberInput">

<!-- Button to trigger the function -->

<button onclick="checkEvenOrOdd()">Check</button>

<!-- Paragraph element to display the result -->

<p id="resultParagraph"></p>

</body>

</html>

**Part 3: Essay Question (40 points)**

Discuss the importance of object-oriented programming (OOP) concepts in software development. Explain the key principles of OOP (encapsulation, inheritance, polymorphism, abstraction) and provide examples of how they can be used to create more efficient, maintainable, and reusable code. Include real-world scenarios or cases where OOP is particularly valuable.

Points Distribution:

Each part carries equal weight (30 points).

Code clarity, functionality, and explanations will be considered in grading.

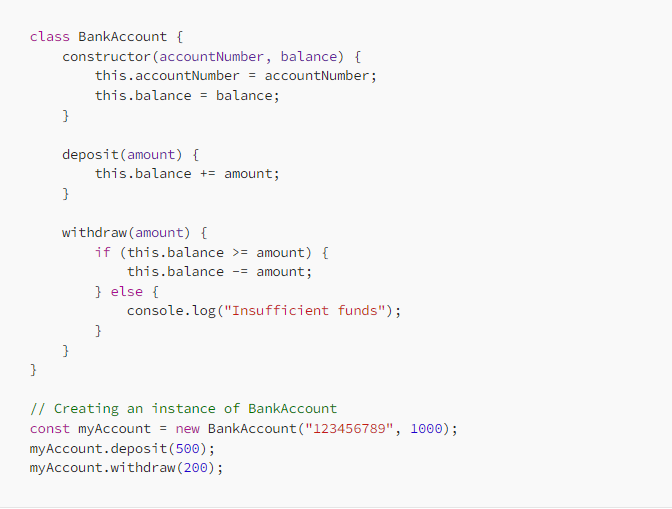
The essay question focuses on understanding and application of OOP concepts.

**Answer:**

Object-Oriented Programming (OOP) stands as a fundamental paradigm in the world of software development, revolutionizing the way programmers approach problem-solving and software design. At its simplest, Object-Oriented Programming can be defined as a programming paradigm that models real-world entities and their interactions through the creation and manipulation of objects. These objects are instances of classes, which act as blueprints or templates for creating objects. OOP promotes the idea of breaking down complex problems into manageable, modular components, making it easier to design, implement, and maintain software.

OOP has become the backbone of modern software development due to its numerous advantages. It fosters code reusability, enabling developers to create libraries of classes and objects that can be employed in various projects. This not only speeds up development but also reduces the likelihood of errors since well-tested components can be reused. Moreover, OOP enhances the scalability of projects, making them more adaptable to changing requirements. Collaboration among developers is also streamlined, as objects encapsulate data and behavior, providing a clear interface for communication.

Object-Oriented Programming (OOP) is founded on a set of core principles that shape the design and construction of software systems. These principles — encapsulation, inheritance, abstraction, and polymorphism — serve as the pillars of OOP, offering a framework for crafting modular, adaptable, and maintainable code. First, encapsulation involves bundling data (properties) and the functions (methods) that manipulate that data into a single unit known as a class. This unit enforces controlled access to the data, allowing external entities to interact with the object’s state only through designated methods. Visualize a TV remote as an encapsulated object. The remote conceals its internal circuitry and buttons. Users can interact with the remote through its exposed buttons, but they don’t need to understand the complex electronics inside. This encapsulation promotes data integrity and prevents unintended data modification, a fundamental aspect of OOP.



Example code for Encapsulation.

**Class Definition** (*BankAccount*):

The *BankAccount* class encapsulates the attributes and methods related to a bank account. *accountNumber* and balance are private attributes encapsulated within the class. They are accessed and modified through the class's methods, not directly from outside the class.

**Constructor**:

The constructor method *(constructor(accountNumber, balance))* is responsible for initializing the *accountNumber* and balance attributes when an instance of *BankAccount* is created. The constructor encapsulates the initialization process.

**Methods** (*deposit* and *withdraw*):

The deposit and withdraw methods are responsible for modifying the balance attribute. These methods encapsulate the logic for depositing and withdrawing funds.

The withdraw method encapsulates the logic to ensure that a withdrawal cannot occur if the account balance is insufficient. It displays "Insufficient funds" when necessary.

**Object Creation** (*myAccount*):

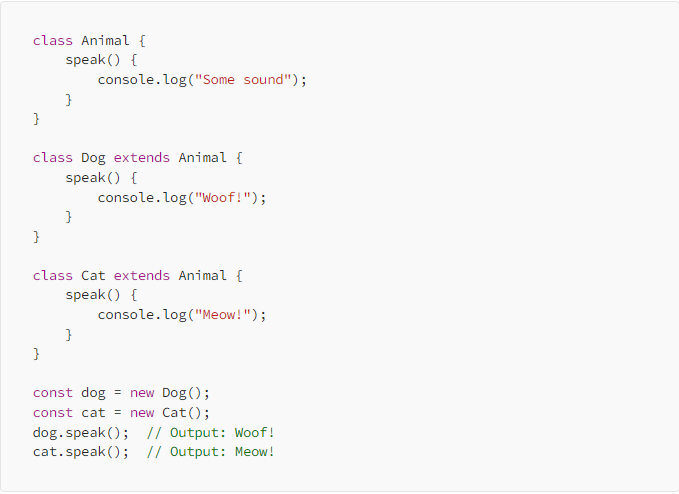
When we create an instance of the *BankAccount* class with const *myAccount* = *new* *BankAccount("123456789", 1000);*, we encapsulate the account's data (*accountNumber* and *balance*) within this object.

**Interaction with Encapsulated Data**:

We interact with the *myAccount* object's data (e.g., depositing and withdrawing funds) through its methods (*deposit* and *withdraw*), ensuring that the data remains encapsulated and controlled by the class.

The code demonstrates encapsulation by encapsulating the attributes and methods related to a bank account within the *BankAccount* class.

The external world interacts with the bank account object (*myAccount*) only through the provided methods (*deposit* and *withdraw*), ensuring that the internal state of the object (*balance*) is protected and manipulated in a controlled manner.

Second, inheritance enables the creation of new classes (subclasses) that inherit attributes and methods from existing classes (superclasses). This promotes code reuse, allowing you to extend or modify the behavior of the superclass without duplicating code. Consider the animal kingdom. Animals share common characteristics, such as movement and reproduction. Inheritance allows us to create specialized classes like “Mammal” and “Bird,” inheriting the general attributes from an overarching “Animal” class. Each subclass customizes the inherited method to exhibit its own behavior. Inheritance allows for code reuse, promoting a hierarchical structure in your code, and enabling you to model relationships between classes in a more organized and efficient manner.

Example code for Inheritance.

**Base Class** (*Animal*):

The *Animal* class is the base class or superclass. It defines a method called *speak*(), which logs "Some sound" to the console. This method is inherited by its subclasses.

Subclasses (*Dog* and *Cat*):

The *Dog* and *Cat* classes are subclasses or derived classes. They extend the Animal class, inheriting its *speak*() method.

**Method Overriding**:

Both *Dog* and *Cat* classes override the *speak*() method. This means they provide their own implementation of the *speak*() method, replacing the one inherited from Animal.

Object Creation (*dog* and *cat*):

We create instances of the *Dog* and *Cat* classes using *const dog = new Dog();* and *const cat = new Cat();*. These objects have access to the *speak*() method due to inheritance.

Method Invocation *(dog.speak()* and *cat.speak())*:

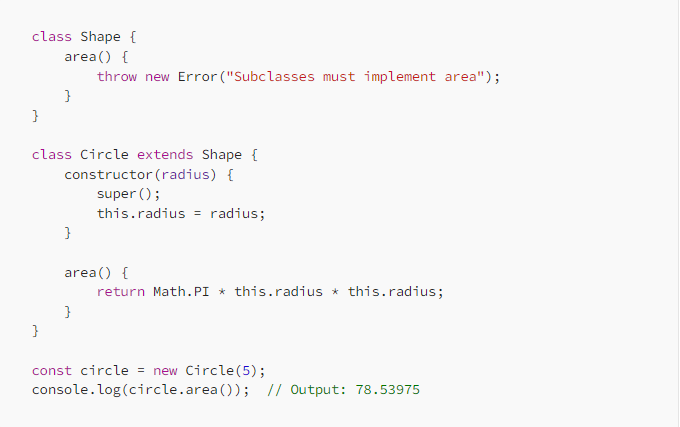
When we call *dog.speak*(), it invokes the *speak*() method of the *Dog* class, and "*Woof!*" is logged to the console. When we call *cat.speak*(), it invokes the *speak*() method of the *Cat* class, and "*Meow*!" is logged to the console.

**Inheritance Relationship**:

*Dog* and *Cat* inherit the *speak*() method from the *Animal* class. This represents an is-a relationship, where a *Dog* is an Animal, and a *Cat* is an *Animal*.

Despite the inheritance, each subclass provides its own unique implementation of the *speak*() method. This is called method overriding, where a subclass provides a specialized implementation of a method inherited from the superclass.

In summary, this code demonstrates inheritance by creating subclasses (*Dog* and *Cat*) that inherit a common method (*speak*()) from the base class (*Animal*).

Third, abstraction focuses on highlighting essential aspects of an object while hiding irrelevant details. Abstract classes and interfaces define a contract that concrete classes must adhere to. This enables high-level design without getting bogged down in implementation specifics. Picture a vending machine. Users select products without knowing the machine’s internal mechanics. The machine abstracts the complexity behind a user-friendly interface.

Example code for Abstraction.

**Abstract Base Class** (*Shape*):

The *Shape* class serves as an abstract base class. It defines an abstract method called *area*(). An abstract method is a method declared in the base class but without an implementation. In this case, *area*() is defined to throw an error message indicating that subclasses must implement this method.

**Concrete Subclass** (*Circle*):

The *Circle* class is a concrete subclass that extends the Shape class. It inherits the *area*() method from the *Shape* class but provides its own implementation. The *constructor(radius)* method initializes the radius property specific to circles.

**Method Implementation** (*area*() in *Circle*):

In the *Circle* class, the *area*() method is implemented with a formula to calculate the area of a circle using the provided radius. This implementation is specific to circles.

**Object Creation** (circle):

We create an instance of the Circle class using *const circle = new Circle(5);*. This object encapsulates a *circle* with a radius of 5.

**Method Invocation** (*circle.area()*):

When we call *circle.area*(), it invokes the *area*() method of the Circle class, which calculates and returns the area of the circle.

**Abstraction in Action**:

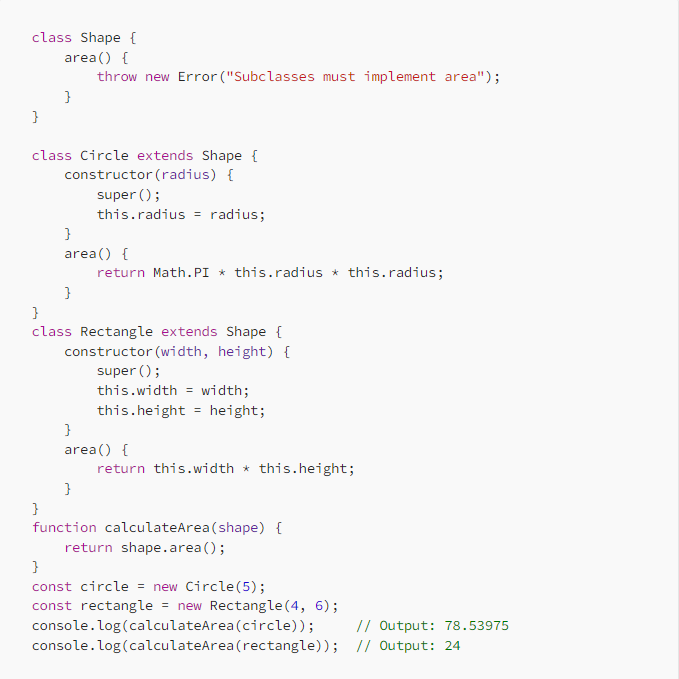
The *Shape* class serves as an abstraction because it defines a common interface (*area*()) for all shapes without specifying how each shape should calculate its area.

Concrete subclasses like *Circle* must implement the abstract method *area*(). This enforces a contract that all shape classes must provide their own implementation of the *area*() method.

In this way, abstraction allows us to define a common structure for shape classes while leaving the specific implementation details to individual shape subclasses. It simplifies the design by focusing on essential characteristics shared by all shapes while hiding the complex math involved in calculating each shape’s area.

In summary, this code demonstrates abstraction by defining an abstract base class (*Shape*) with an abstract method (*area*) and a concrete subclass (*Circle*) that implements this method. Abstraction simplifies the process of modeling and working with complex systems, making code more maintainable and extensible.

And lastly, polymorphism allows objects of different classes to be treated as objects of a common superclass. This enables writing flexible, generalized code that can work with various object types, leveraging method overriding and interfaces. Consider a “Shape” class. Polymorphism lets you calculate areas of different shapes — circles, rectangles, and triangles—using a common method, even though their implementations differ.



Example code for Polymorphism.

**Base Class** (*Shape*):

The *Shape* class serves as the base class. It defines an abstract method called *area*(). This method is marked as abstract by throwing an error, indicating that subclasses must implement it.

**Concrete Subclasses** (*Circle* and *Rectangle*):

The *Circle* and *Rectangle* classes are concrete subclasses of Shape. They inherit the *area*() method from Shape but provide their own implementations.

The *Circle* class calculates the area of a circle based on its radius, while the *Rectangle* class calculates the area of a rectangle based on its *width* and *height*.

**Polymorphism in** *calculateArea*():

The *calculateArea*(*shape*) function demonstrates polymorphism. It takes an object of type Shape (or any subclass of *Shape*) as its parameter. Since both *Circle* and *Rectangle* are subclasses of *Shape*, they can be passed as arguments to this function.

**Method Invocation** (*shape.area*()):

Inside *calculateArea*(), the *area*() method is called on the shape object. Depending on whether shape is a *Circle* or *Rectangle*, the appropriate *area*() method is executed, demonstrating polymorphic behavior.

**Object Creation** (circle and rectangle):

We create instances of *Circle* and *Rectangle* classes using *const circle = new Circle(5);* and *const rectangle = new Rectangle(4, 6);*, respectively.

**Calculating Areas:**

When we call *calculateArea*(*circle*), it calculates the area of the circle using the *area*() method implemented in the *Circle* class.

When we call *calculateArea*(*rectangle*), it calculates the area of the *rectangle* using the *area*() method implemented in the *Rectangle* class.

**Polymorphism in Action:**

Polymorphism allows us to write a single function (*calculateArea*) that can work with different shapes without knowing their specific types.

By defining a common interface (*area*()) in the base class (*Shape*) and implementing it differently in subclasses (*Circle* and *Rectangle*), we achieve dynamic behavior based on the actual type of the object passed.

In short, this code demonstrates polymorphism by allowing objects of different classes (*Circle* and *Rectangle*) to be treated as objects of a common base class (*Shape*). The function *calculateArea*() showcases how polymorphism enables dynamic behavior based on the actual type of the object, promoting flexibility and code reusability.

In summary, OOP introduces a novel way of structuring code by organizing data and its related functions into cohesive units called “objects” and founded on a set of core principles that shape the design and construction of software system. These principles -- encapsulation, inheritance, abstraction, and polymorphism empower you to design and implement intricate systems in a structured and efficient manner.