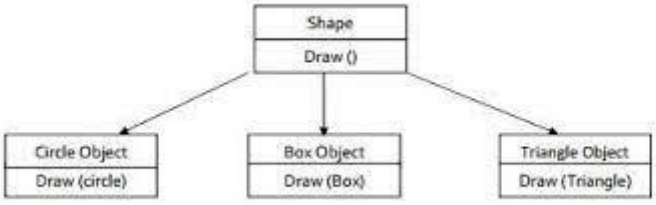
SCT 121-0941/2022

VYONA NJERI

Part A

i)



Identify the objects,define classes ,establish relationships.

ii) The Object Modeling Technique (OMT) is an object-oriented analysis, design, and implementation methodology that focuses on creating a model of objects from the real world and then using this model to develop object-oriented software.

iii) OOAD is a software engineering methodology that involves using object-oriented concepts to design and implement software systems, while OOP is a programming paradigm that uses objects to represent real-world entities and involves creating objects that have properties and methods, and using those objects to build applications.

iv) **Standardization**-Provide a standardized way to visualize the design of a system.

Constructive modelling-support forward and reverse engineering activities.

To provide a common language- that can be used to model a system's structure and behaviour.

V) **modularity** – Oop allows for the modular organization of code ,making it easier to understand , maintain and update.

Reausability- objects and classes can be reused in different parts of the system or in other projects saving time and effort.

Encapsulation- The data within an object is kept private, ensuring that the data is protected from being accidentally altered or accessed.

vi) A **constructor** is a special type of function called to create an object.They have the same name as the class and is used to initialize the object's attributes.

Example:

```
public class MyClass {
    int x;
```

```
    public MyClass(int y) {
        x = y;
    }
}
```

An **object** is an instance of a class. It has its own attributes and behaviours.

```
public class MyClass {
    int x;

    public MyClass(int y) {
        x = y;
    }

    public void printX() {
        System.out.println("x = " + x);
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass myObject = new MyClass(5);
        myObject.printX(); // Output:
```

A **destructor** is an instance member function that is invoked automatically whenever an object is going to be destroyed.Java doesn't have explicit destructors as it has automatic garbage collection.

Polymorphism allows objects to be treated as instances of their base class.It includes overloading and overriding.

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }
```

```
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape myShape = new Circle(); // Polymorphism
        myShape.draw(); // Calls the draw method of Circle
    }
}
```

Class- A blueprint for creating objects. It defines the states and behaviour of an object.

```
public class Dog {
    // Attributes
    String name;
    int age;

    // Behavior
    void bark() {
        System.out.println("Woof!");
    }
}
```

Inheritance- The ability of a subclass to inherit attributes and behaviours of another class (super class).

```
class Animal {
    void eat() {
        System.out.println("Eating");
```

```
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat(); // Inherited from Animal
        myDog.bark(); // From Dog class
    }
}
```

vi)Association- It represents a connection between two or more objects without implying any form of ownership

Aggregation- it is a relationship between two classes where one class has a reference to one or more instances of another class.

Composition- Composition is a relationship between two classes where one class is composed of one or more instances of another class.

Generalisation-They are used to display an inheritance relationship between the two classes.

vii) A **classdiagram** is a type of UML diagram that represents the structure and relationships of classes within a system. Classdiagrams are used in software development to illustrate the static structure of a system.

Steps to draw a class diagram

Identify classes

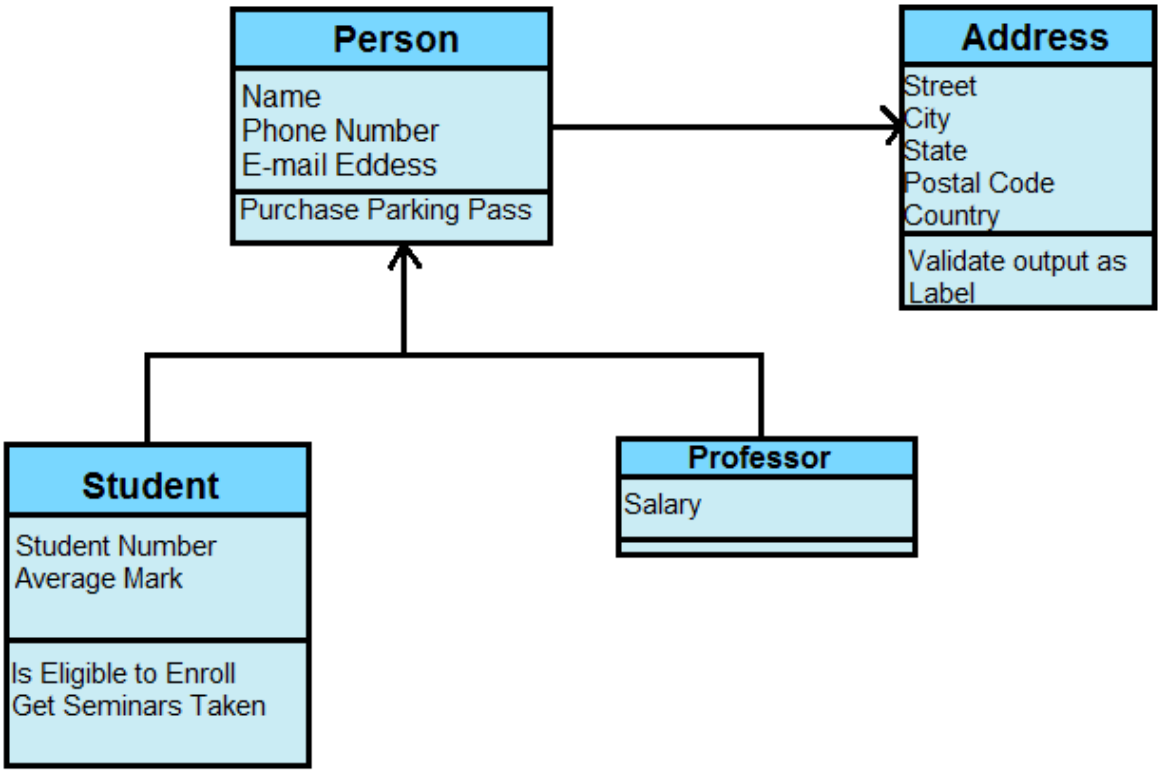
Identify attributes

Identify methods

Identify relationships

Add inheritance if needed.

Example



vii) *#include <iostream>*

#include <cmath>

using namespace std;

```
class Shape {
public:
    virtual float area() = 0;
    virtual float perimeter() = 0;
};

class Circle : public Shape {
private:
    float radius;
```

```
public:
    Circle(float r) {
        radius = r;
    }
    float area() {
        return M_PI * pow(radius, 2);
    }
    float perimeter() {
        return 2 * M_PI * radius;
    }
    friend void display(Circle c);
};

class Rectangle : public Shape {
private:
    float length, width;
public:
    Rectangle(float l, float w) {
        length = l;
        width = w;
    }
    float area() {
        return length * width;
    }
    float perimeter() {
        return 2 * (length + width);
    }
    friend void display(Rectangle r);
};

class Triangle : public Shape {
private:
```

```
    float a, b, c;
public:
    Triangle(float x, float y, float z) {
        a = x;
        b = y;
        c = z;
    }
    float area() {
        float s = (a + b + c) / 2;
        return sqrt(s * (s - a) * (s - b) * (s - c));
    }
    float perimeter() {
        return a + b + c;
    }
    friend void display(Triangle t);
};

class Square : public Shape {
private:
    float side;
public:
    Square(float s) {
        side = s;
    }
    float area() {
        return pow(side, 2);
    }
    float perimeter() {
        return 4 * side;
    }
    friend void display(Square s);
};
```

```
void display(Circle c) {
    cout << "Area of circle: " << c.area() << endl;
    cout << "Perimeter of circle: " << c.perimeter() << endl;
}

void display(Rectangle r) {
    cout << "Area of rectangle: " << r.area() << endl;
    cout << "Perimeter of rectangle: " << r.perimeter() << endl;
}

void display(Triangle t) {
    cout << "Area of triangle: " << t.area() << endl;
    cout << "Perimeter of triangle: " << t.perimeter() << endl;
}

void display(Square s) {
    cout << "Area of square: " << s.area() << endl;
    cout << "Perimeter of square: " << s.perimeter() << endl;
}

int main() {
    Circle c(5);
    Rectangle r(5, 6);
    Triangle t(3, 4, 5);
    Square s(7);

    display(c);
    display(r);
    display(t);
    display(s);
```

```
    return 0;
}
```

a. **Inheritance**- The ability of a subclass to inherit attributes and behaviours of another class (super class).

In this implementation, we use single inheritance to create a base class called `Shape` and derive four classes from it: `Circle`, `Rectangle`, `Triangle`, and `Square`. Each derived class inherits the properties and methods of the `Shape` class.

b. **Friends functions**- They are non-member functions that are granted access to the private and protected members of a class. In this implementation, we use a friend function called `display` to display the area and perimeter of each shape.
c. **Method overloading and method overriding:-** Method overloading is a feature that allows multiple functions with the same name to be defined in a class. Method overriding is a feature that allows a subclass to provide a specific implementation of a method that is already provided by its parent class .

In this implementation, we use method overloading to define multiple constructors for each derived class, and we use method overriding to override the `display` method in each derived class.

d. **Late binding and early binding-** Early binding is a process in which the compiler associates an address to a function call at compile time. Late binding is a process in which the address of a function call is resolved at runtime.

In this implementation, we use late binding to resolve the `display` method at runtime.

e. **Abstract class and pure functions -** An abstract class is a class that cannot be instantiated and is used as a base class for other classes . A pure function is a virtual function that is set to 0 and has no implementation. In this implementation, we use an abstract class called `Shape` to define pure virtual functions for calculating the area and perimeter of each shape.

viii) a. **Function overloading example**

```
#include <iostream>
using namespace std;

int add(int x, int y) {
    return x + y;
}

double add(double x, double y) {
    return x + y;
}

int main() {
    cout << add(2, 3) << endl;
    cout << add(2.0, 3.0) << endl;
    return 0;
}
```

a. Operator overloading example

```
#include <iostream>
using namespace std;

class Complex {
public:
    int real, imag;
    Complex operator+(Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
};

int main() {
    Complex c1, c2, c3;
    c1.real = 2;
    c1.imag = 3;
    c2.real = 4;
    c2.imag = 5;
    c3 = c1 + c2;
    cout << c3.real << " + i" << c3.imag << endl;
    return 0;
}
```

b. Pass by value and pass by reference

```
#include <iostream>

void increment(int value) { // pass by value
    value++;
}

void increment_ref(int& value) { // pass by reference
    value++;
}

int main() {
    int x = 5;
    std::cout << "Original value of x: " << x << std::endl;
    increment(x);
    std::cout << "After increment(x): " << x << std::endl;
    increment_ref(x);
    std::cout << "After increment_ref(x): " << x << std::endl;
    return 0;
}
```

In the above example, the " increament" function uses pass by value, while the "increament ref" function uses pass by reference.

c. Parameters and arguments

```
#include <iostream>

void print(int value) {
   std::cout << "Printing int: " << value << std::endl;
}

int main() {
   int x = 42;
   print(x);
   return 0;
}
```

A parameter is a variable in a function's declaration that specifies the type of argument the function should accept. In this case, the parameter **'value'** is of type int.

An argument is a value that is passed into a function when it is called. In this case, the argument **'x'** is of type int The value of x is passed into the function print , and the function prints the value of the argument.

Here is the modified version of the `CalculateG` class that computes the position and velocity of an object after falling for 30 seconds, outputting the position in meters:

```
 #include <iostream>
#include <cmath>

using namespace std;

class CalculateG {
public:
   double gravity = -9.81; // Earth's gravity in m/s^2
   double fallingTime = 30;
   double initialVelocity = 0.0;
   double finalVelocity = initialVelocity + gravity * fallingTime;
   double initialPosition = 0.0;
   double finalPosition = 0.5 * gravity * pow(fallingTime, 2) + initialVelocity *
fallingTime + initialPosition;
   void outline() {
      cout << "The object's position after " << fallingTime << " seconds is " <<
finalPosition << " m." << endl;
      cout << "The object's velocity after " << fallingTime << " seconds is " <<
finalVelocity << " m/s." << endl;
   }
};

int main() {
   CalculateG calc;
   calc.outline();
   return 0;
```

```
}
```

Here is the extended `CalculateG` class with the methods for multiplication, powering to square, summation, and printing out a result:

```
#include <iostream>
#include <cmath>

using namespace std;

class CalculateG {
public:
    double multi(double a, double b) {
        return a * b;
    }

    double power(double a) {
        return pow(a, 2);
    }

    double sum(double a, double b) {
        return a + b;
    }

    void outline(double result) {
        cout << "The result is " << result << endl;
    }

    double gravity = -9.81; // Earth's gravity in m/s^2
    double fallingTime = 30;
    double initialVelocity = 0.0;
    double finalVelocity = initialVelocity + gravity * fallingTime;
    double initialPosition = 0.0;
    double finalPosition = 0.5 * gravity * pow(fallingTime, 2) + initialVelocity
* fallingTime + initialPosition;

    void outline() {
        cout << "The object's position after " << fallingTime << " seconds is "
<< finalPosition << " m." << endl;
        cout << "The object's velocity after " << fallingTime << " seconds is "
<< finalVelocity << " m/s." << endl;
    }
};

int main() {
    CalculateG calc;
    double a = 2.0;
    double b = 3.0;
```

```
    double result1 = calc.multi(a, b);
    double result2 = calc.power(a);
    double result3 = calc.sum(a, b);
    calc.outline(result1);
    calc.outline(result2);
    calc.outline(result3);
    calc.outline();
    return 0;
}
```

PART B

Question 1

```
#include <iostream>

using namespace std;

int main() {
    int a = 1, b = 2, c = 0, sum = 2;
    while (c <= 4000000) {
        c = a + b;
        if (c % 2 == 0) {
            sum += c;
        }
        a = b;
        b = c;
    }
    cout << "The sum of all even-valued terms in the Fibonacci sequence whose values do not exceed four million is " << sum << endl;
    return 0;
}
```

QUESTION THREE

```
#include <iostream>

using namespace std;
```

```
int main() {
    int arr[15];
    for (int i = 0; i < 15; i++) {
        cout << "Enter value " << i + 1 << ": ";
        cin >> arr[i];
    }
    cout << "The values stored in the array are: ";
    for (int i = 0; i < 15; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    int num;
    cout << "Enter a number to search for: ";
    cin >> num;
    bool found = false;
    int index;
    for (int i = 0; i < 15; i++) {
        if (arr[i] == num) {
            found = true;
            index = i;
            break;
        }
    }
    if (found) {
        cout << "The number " << num << " was found
at index " << index << "." << endl;
    } else {
        cout << "The number " << num << " was not
found in this array." << endl;
    }
    int arr2[15];
    for (int i = 0; i < 15; i++) {
        arr2[i] = arr[14 - i];
    }
```

```
    cout << "The values stored in the new array are: ";
    for (int i = 0; i < 15; i++) {
        cout << arr2[i] << " ";
    }
    cout << endl;
    int sum = 0;
    int product = 1;
    for (int i = 0; i < 15; i++) {
        sum += arr[i];
        product *= arr[i];
    }
    cout << "The sum of all elements of the array is "
<< sum << "." << endl;
    cout << "The product of all elements of the array is
" << product << "." << endl;
    return 0;
}
```