DIT: OOP Assignment

SCT121-0867/2022

Alfred Mutua

Part A:

Instructions for part A: answer all the Questions in this section.

i. Using a well labeled diagram, explain the steps of creating a system using OOP principles. [4 Marks]

1. **Identify the objects**: The first step is to identify the objects that will be used in the system. Objects are instances of classes and represent real-world entities. For example, in a library management system, objects could be books, authors, and borrowers.

2. **Define the classes**: Once the objects have been identified, the next step is to define the classes. A class is a blueprint for creating objects. It defines the attributes and methods that an object will have. For example, a book class could have attributes like title, author, and ISBN, and methods like borrow and return.

3. **Create the relationships between classes**: After defining the classes, the next step is to create the relationships between them. There are three types of relationships between classes: inheritance, composition, and aggregation. Inheritance is when a class inherits attributes and methods from another class. Composition is when a class is made up of other classes. Aggregation is when a class has a reference to another class.

4. **Create the object diagram**: An object diagram is a visual representation of the objects and their relationships. It shows how the objects interact with each other. For example, in a library management system, an object diagram could show how a borrower borrows a book from the library.

5. **Implement the classes**: The final step is to implement the classes. This involves writing the code for the classes and their methods. Once the classes have been implemented, objects can be created and used in the system.

ii. What is the Object Modeling Techniques (OMT). [1 Marks]
The Object Modeling Technique (OMT) is a methodology for object-oriented analysis and design (OOAD). It provides a structured approach to breaking down real-world problems into manageable pieces and then translating them into object-oriented software.

iii. Compare object-oriented analysis and design (OOAD) and object analysis and design (OOP). [2 Marks]

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 design applications and computer programs.

iv. Discuss Main goals of UML. [2 Marks]

UML's main goals in OOP are to facilitate communication, guide design, promote abstraction and reusability, support analysis and testing, ensure platform independence, manage complexity, and integrate with development tools.

v. DESCRIBE three advantages of using object oriented to develop an information system. [3Marks]

- **Modularity:** OOP allows developers to break down a large problem into smaller, more manageable pieces. Each piece can be developed and tested independently, which makes it easier to troubleshoot problems when they arise. This modularity also allows multiple developers to work on different parts of the system simultaneously, without interfering with each other's work.

- **Code Reusability:** OOP promotes code reuse by allowing developers to create classes that can be used in multiple projects. This saves time and effort, as developers don't have to write the same code over and over again. In addition, OOP allows developers to inherit properties and methods from existing classes, which can further reduce development time.

- **Flexibility:** OOP allows developers to create complex systems that can be easily modified and extended. This is because OOP is based on the concept of objects, which can be modified and extended without affecting the rest of the system. This makes it easier to add new features to an existing system, or to modify existing features without breaking the system.

vi. Briefly explain the following terms as used in object-oriented programming. Write a sample java code to illustrate the implementation of each concept. [12 Marks]

a. Constructor

A constructor is a special method in object-oriented programming that is called when an object is created.

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

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

b. object

An object refers to an instance of a class.

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

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

    public void sayHello() {
        System.out.println("Hello, my name is " + name + " and I am " + age + " years old.");
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("John", 30);
        person.sayHello();
    }
}
```

```
class Resource {
  private File file;

  public Resource(String filePath) {
    this.file = new File(filePath);
  }

  @Override
  public void finalize() throws Throwable {
    super.finalize();
    if (file != null && file.isOpened()) {
      file.close();
      System.out.println("Resource cleaned up: " + file.getName());
    }
  }
}

public class Main {
  public static void main(String[] args) {
    Resource resource = new Resource("data.txt");
    // ... use the resource ...
    resource = null; // resource becomes unreachable
  }
}
```

a. polymorphism

Polymorphism, in object-oriented programming (OOP), refers to the ability of an object to take on multiple forms.

```
class Animal {
  public void speak() {
    System.out.println("I am an animal.");
  }
}

class Dog extends Animal {
  @Override // Overriding speak method from Animal
  public void speak() {
    System.out.println("Woof!");
  }

  public void bark() {
    System.out.println("Bark, bark!");
  }
}
public class Main {
  public static void main(String[] args) {
    Animal animal = new Animal ();
    animal.speak(); // Outputs: "I am an animal."

    Dog dog = new Dog ();
    dog.speak(); // Outputs: "Woof!" (Overriding behavior)
    dog.bark(); // Outputs: "Bark, bark!" (Overloading)
```

b. class

A class in OOP acts as a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that class will share.

```
public class Person {
 // Define attributes (properties)
 private String name;
 private int age;
 private String address;
 // Define methods (behaviors)
 public void setName(String name) {
  this.name = name;
 }
 public String getName() {
  return name;
 }
 public void setAge(int age) {
  this.age = age;
 }
 public int getAge() {
  return age;
 }
 public void setAddress(String address) {
  this.address = address;
 }
 public String getAddress() {
  return address;
 }
```

c. Inheritance

This refers to the ability of an object to acquire all methods and properties of a parent class.

```
public class Circle extends Shape {

 private double radius;



 public Circle(double radius) {

  this.radius = radius;

 }



 @Override

 public double getArea() {

  return Math.PI * radius * radius;

 }



 public void setRadius(double radius) {

  this.radius = radius;

 }



 public double getRadius() {

  return radius;

 }



 @Override

 public String toString() {

  return "Circle [color=" + color + ", radius=" + radius + "]";

 }

}
```

vi. *EXPLAIN* the three types of associations (relationships) between objects in object oriented. [6 Marks]

1. **Association**: Association is a relationship between two separate classes that establishes through their objects. It can be one-to-one, one-to-many, many-to-one, or many-to-many.
2. **Aggregation**: Aggregation is a special form of association where one class is a part of another class. It represents a "has-a" relationship, where one class contains a reference to another class.
3. **Composition**: Composition is a stronger form of aggregation where one class is a part of another class, but the contained class cannot exist without the container class.

Vii. What do you mean by class diagram? Where it is used and also discuss the steps to draw the class diagram with any one example. [6 Marks]

1. **Identify the classes:** The first step is to identify the classes in the system. Classes are usually nouns that represent objects in the system.
2. **Determine the relationships between classes:** The next step is to determine the relationships between the classes. There are three types of relationships between classes: association, aggregation, and composition.
3. **Add attributes and operations:** Once the classes and relationships have been identified, the next step is to add attributes and operations to the classes. Attributes are the data members of a class, while operations are the methods that can be performed on the class.
4. **Add multiplicity:** Multiplicity is used to indicate the number of instances of one class that are linked to one instance of another class. It is represented by a number or a range of numbers.
5. **Add constraints:** Constraints are used to specify additional requirements or conditions that must be met by the system. They can be added to the class diagram using OCL (Object Constraint Language).

ii. Given that you are creating area and perimeter calculator using C++, to compute area and perimeter of various shaped like Circles, Rectangle, Triangle and Square, use well written code to explain and implement the calculator using the following OOP concepts**.**

a. Inheritance (Single inheritance, Multiple inheritance and Hierarchical inheritance)

Single Inheritance

```
#include <iostream>
#include <cmath>
using namespace sty;
// Base class
class Shape {
public:
   virtual double getArea() = 0;
   virtual double getPerimeter() = 0;
};
// Derived class for Circle
class Circle : public Shape {
private:
   double radius;
public:
   Circle(double r) {
      radius = r;
   }
   double getArea() {
      return M_PI * pow(radius, 2);
   }
   double getPerimeter() {
      return 2 * M_PI * radius;
   }
};
// Derived class for Rectangle
class Rectangle : public Shape {
private:
   double length;
   double width;
```

Multiple Inheritance

```
#include <iostream>
#include <cmath>
using namespace std;
// Base class
class Shape {
public:
   virtual double getArea() = 0;
   virtual double getPerimeter() = 0;
};
// Derived class for Circle
class Circle : public Shape {
private:
   double radius;
public:
   Circle(double r) {
      radius = r;
   }
   double getArea() {
      return M_PI * pow(radius, 2);
   }
   double getPerimeter() {
      return 2 * M_PI * radius;
   }
};
// Derived class for Rectangle
class Rectangle : public Shape {
```

Hierarchical Inheritance

```
#include <iostream>
#include <cmath>
using namespace std;
// Base class
class Shape {
public:
   virtual double getArea() = 0;
   virtual double getPerimeter() = 0;
};
// Derived class for Circle
class Circle : public Shape {
private:
   double radius;
public:
   Circle(double r) {
      radius = r;
   }
   double getArea() {
      return M_PI * pow(radius, 2);
   }
   double getPerimeter() {
      return 2 * M_PI * radius;
   }
};
```

b. Friend functions [5 Marks]

```
#include <iostream>
#include <cmath>
using namespace std;
// Forward declaration of classes
class Circle;
class Rectangle;
class Triangle;
class Square;
// Base class
class Shape {
public:
   virtual double getArea() = 0;
   virtual double getPerimeter() = 0;
};
// Derived class for Circle
class Circle : public Shape {
private:
   double radius;
public:
   Circle(double r) {
      radius = r;
   }
   double getArea() {
      return M_PI * pow(radius, 2);
   }
   double getPerimeter() {
```

c. Method overloading [10 Marks] 1

Method overloading is a powerful feature in object-oriented programming (OOP) that allows a class to define multiple methods with the same name, but with different functionalities.

```
#include <iostream>

#include <cmath>



using namespace std;



// Base class

class Shape {

public:

   virtual double getArea() = 0;

   virtual double getPerimeter() = 0;

};



// Derived class for Circle

class Circle : public Shape {

private:

   double radius;

public:

   Circle(double r) {

      radius = r;

   }

   double getArea() {

      return M_PI * pow(radius, 2);

   }

   double getPerimeter() {
```

Method Overriding

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

using namespace std;

// Base class
class Shape {
public:
    virtual double getArea() = 0;
    virtual double getPerimeter() = 0;
};

// Derived class for Circle
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) {
        radius = r;
    }
    double getArea() override {
        return M_PI * pow(radius, 2);
    }
    double getPerimeter() override {
        return 2 * M_PI * radius;
```

d. Late binding

Late binding refers to the process of determining the implementation of a method at runtime, rather than at compile time.

```
#include <iostream>

#include <cmath>



using namespace std;



// Base class

class Shape {

public:

   virtual double getArea() = 0;

   virtual double getPerimeter() = 0;

};



// Derived class for Circle

class Circle : public Shape {

private:

   double radius;

public:

   Circle(double r) {

      radius = r;

   }

   double getArea() {

      return M_PI * pow(radius, 2);

   }

   double getPerimeter() {

      return 2 * M_PI * radius;
```

Early binding

Refers to the process of determining the implementation of a method at compile time.

```
#include <iostream>

#include <cmath>

class Shape {

public:

   // Early binding: Determine area and perimeter at compile time based on object type

   double area() const {

      if (dynamic_cast<Circle*>(this)) return getCircleArea(); // Early binding for Circle

      else if (dynamic_cast<Rectangle*>(this)) return getRectangleArea(); // Early binding for Rectangle

      else if (dynamic_cast<Triangle*>(this)) return getTriangleArea(); // Early binding for Triangle

      else if (dynamic_cast<Square*>(this)) return getSquareArea(); // Early binding for Square

      else throw std::runtime_error("Unknown shape"); // Handle unsupported shapes

   }

   double perimeter() const {

      if (dynamic_cast<Circle*>(this)) return getCirclePerimeter(); // Early binding for Circle

      else if (dynamic_cast<Rectangle*>(this)) return getRectanglePerimeter(); // Early binding for Rectangle

      else if (dynamic_cast<Triangle*>(this)) return getTrianglePerimeter(); // Early binding for Triangle

      else if (dynamic_cast<Square*>(this)) return getSquarePerimeter(); // Early binding for Square

      else throw std::runtime_error("Unknown shape"); // Handle unsupported shapes

   }

   void printInfo() const {

      std::cout << "Shape: " << typeid(*this).name() << ", Area = " << area() << ", Perimeter = " << perimeter() <<
std::endl;

   }

protected: // Can be accessed by derived classes

   double getCircleArea() const { return 0.0; } // Pure virtual for Circle
```

e. Abstract class [6 Marks]

They define the core functionality while allowing individual classes to specialize in their specific formulas and behavior.

```
#include <iostream>

#include <cmath>



class Shape {

public:

   virtual double area() const = 0; // Pure virtual function for area

   virtual double perimeter() const = 0; // Pure virtual function for perimeter

   virtual void printInfo() const {

      std::cout << "Shape: " << typeid(*this).name() << std::endl; // Print type name

   }

};
```

Pure functions
They enable derived classes to handle their specific calculations while providing a common interface and functionalities in the base class.

```
#include <iostream>

#include <cmath>



class Shape {

public:

 // Pure functions for area and perimeter

 virtual double area(const Shape& shape) const = 0;

 virtual double perimeter(const Shape& shape) const = 0;



 // Optional virtual functions for specific shapes (implement if needed)

 virtual double circleArea(double radius) const = 0;

 virtual double rectangleArea(double length, double width) const = 0;

 virtual double triangleArea(double side1, double side2, double side3) const = 0;

 virtual double squareArea(double side) const = 0;



 // Optional virtual functions for specific shapes (implement if needed)

 virtual double circlePerimeter(double radius) const = 0;

 virtual double rectanglePerimeter(double length, double width) const = 0;

 virtual double trianglePerimeter(double side1, double side2, double side3) const = 0;

 virtual double squarePerimeter(double side) const = 0;



 // Common printInfo function

 void printInfo(const Shape& shape) const {
```

iii. Using a program written in C++, differentiate between the following. [6 Marks]

a. Function overloading

Allows us to define multiple functions with the same name but different parameters in the same scope.

```
#include <iostream>

using namespace std;


void print(int i) {

   cout << "Printing integer: " << i << endl;

}


void print(double  f) {

   cout << "Printing float: " << f << endl;

}


void print(char* c) {

   cout << "Printing character: " << c << endl;

}


int main() {

   print(10);

   print(3.14);

   print("Hello World");

   return 0;

}
```

Operator overloading
That allows us to redefine the meaning of an operator for a user-defined data type.

```
#include <iostream>

using namespace std;


class Complex {

private:

   int real, imag;

public:

   Complex(int r = 0, int i = 0) {

      real = r;

      imag = i;

   }


   Complex operator+(Complex const& obj) {

      Complex res;

      res.real = real + obj.real;

      res.imag = imag + obj.imag;

      return res;

   }


   void print() {

      cout << real << " + i" << imag << endl;

   }

};
```

b. Pass by value and pass by reference **pass by value** and **pass by reference**:

- **Pass by value** is a method of passing arguments to a function by making a copy of the value of the argument. The function works with the copy of the argument, and any changes made to the argument inside the function do not affect the original value of the argument.
- **Pass by reference** is a method of passing arguments to a function by passing the address of the argument instead of the value. The function works with the original argument, and any changes made to the argument inside the function affect the original value of the argument.

c. Parameters and arguments

- **Parameters** are variables declared in the function definition that represent the values passed to the function. They are used to receive the values of the arguments passed to the function.
- **Arguments** are the values passed to a function when it is called. They are used to initialize the parameters of the function.

NOTE: To score high marks, you are required to explain each question in detail. Do good research and cite all the sources of your information. DO NOTE CITE WIKIPEDIA.

Create a new class called *CalculateG.*
Copy and paste the following initial version of the code. Note variables declaration and the types.

```
class CalculateG {
int main(){

(datatype) gravity =-9.81; // Earth's gravity in m/s^2 (datatype) fallingTime = 30;

(datatype)initialVelocity = 0.0; (datatype) finalVelocity = ;

(datatype) initialPosition = 0.0; (datatype) finalPosition = ;
```

```
        // Add the formulas for position and velocity
        Cout<<"The object's position after " << fallingTime << " seconds is "
        + finalPosition + << m."<<endl;
    // Add output line for velocity (similar to position)

} }
```

Modify the example program to compute the position and velocity of an object after falling for 30 seconds, outputting the position in meters. The formula in Math notation is:

$x(t)=0.5*at^2+v_it+x_i\, v(t)=at+v_i$

Run the completed code in Eclipse (Run → Run As → Java Application). 5. Extend *datatype* class with the following code:

```
public class CalculateG {

public double multi(......){ // method for multiplication

}

// add 2 more methods for powering to square and summation (similar to multiplication)

public void outline(......){
// method for printing out a result

}
int main() {

// compute the position and velocity of an object with defined methods and print out
the
result

} }
```

6. Create methods for multiplication, powering to square, summation and printing out a result in *CalculateG* class.

Part B:

Instructions for part B: Do question 1 and any other one question from this section.

1. Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

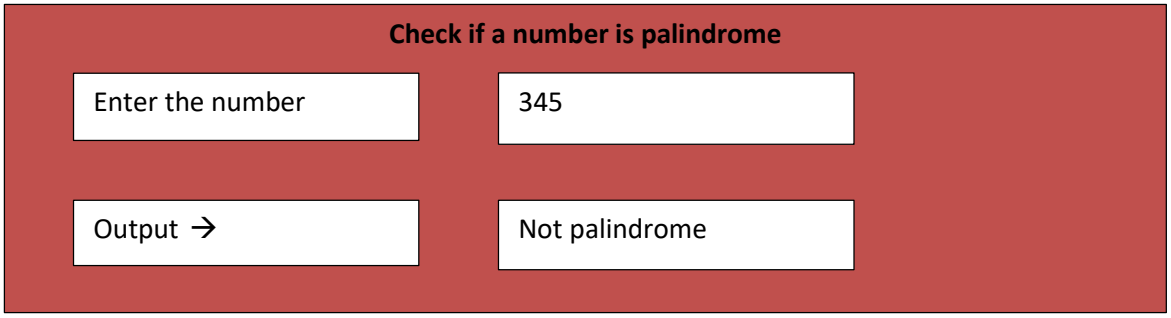
By considering the terms in the Fibonacci sequence whose values do not exceed four million, write a C++ method to find the sum of all the even- valued terms.

```
#include <iostream>

using namespace std;



int main() {

   int limit = 4000000;

   int sum = 0;

   int a = 1, b = 2, c = 0;

   while (b <= limit) {

      if (b % 2 == 0) {

         sum += b;

      }

      c = a + b;

      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 Two: [15 marks]

2. A palindrome number is a number that remain the same when read from behind or front (a number that is equal to reverse of number) for example, 353 is palindrome because reverse of 353 is 353 (you see the number remains the same). But a number like 591 is not palindrome because reverse of 591 is 195 which is not equal to 591. Write C++ program to check if a number entered by the user is palindrome or not. You should provide the user with a GUI interface to enter the number and display the results on the same interface.

The interface:



Question three: [15 marks]

Write a C++ program that takes 15 values of type integer as inputs from user, store the values in an array.

a) Print the values stored in the array on screen.

```
#include <iostream>

using namespace std;


int main() {

   int arr[15];

   cout << "Enter 15 integers:" << endl;

   for (int i = 0; i < 15; i++) {

      cin >> arr[i];

   }

   cout << "The values stored in the array are:" << endl;

   for (int i = 0; i < 15; i++) {

      cout << arr[i] << " ";

   }
```

b) Ask user to enter a number, check if that number (entered by user) is present in array or not. If it is present print, “the number found at index (index of the number) ” and the text “number not found in this array”

```
#include <iostream>

using namespace std;

int main() {

   int arr[15];

   cout << "Enter 15 integers:" << endl;

   for (int i = 0; i < 15; i++) {

      cin >> arr[i];

   }

   int num;

   cout << "Enter a number to search for:" << endl;

   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;

   }

   return 0;
```

c) Create another array, copy all the elements from the existing array to the new array but in reverse order. Now print the elements of the new array on the screen.

```
#include <iostream>

using namespace std;


int main() {

   int arr[15];

   cout << "Enter 15 integers:" << endl;

   for (int i = 0; i < 15; i++) {

      cin >> arr[i];

   }

   int rev_arr[15];

   for (int i = 0; i < 15; i++) {

      rev_arr[i] = arr[14 - i];

   }

   cout << "The elements of the new array in reverse order are:" << endl;

   for (int i = 0; i < 15; i++) {

      cout << rev_arr[i] << " ";

   }

   cout << endl;

   return 0;

}
```

d) Get the sum and product of all elements of your array. Print product and the sum each on its own line.

```
#include <iostream>

using namespace std;


int main() {

   int arr[15];

   cout << "Enter 15 integers:" << endl;

   for (int i = 0; i < 15; i++) {

      cin >> arr[i];

   }

   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;

}
```