EMMANUEL MAGACHI JESSE

SCT 121-0852/2022

OBJECT ORINTED PROGRAMMING ASSINGNMENT

I. USING A WELL LABELED DIAGRAM, EXPLAIN THE STEPS OF CREATING A SYSTEM USING OOP PRINCIPLES.

OOP is a programming paradigm that uses "objects" to design applications and computer programs. It utilizes several key concepts such as encapsulation, inheritance, abstraction, and polymorphism. Below is a description of the steps to create a system using OOP principles, accompanied by a well-labeled diagram that illustrates these steps.

Steps for Creating a System Using OOP Principles:

1. Requirement Analysis:
   - Understand the problem domain.
   - Identify the requirements of the system.
2. Design:
   - Define the system architecture.
   - Create class diagrams to illustrate relationships.
3. Implementation:
   - Write code for the classes and methods.
   - Implement encapsulation to protect data.
4. Testing:
   - Test individual objects.
   - Test the interactions between objects.
5. Deployment:
   - Deploy the system in a production environment.
   - Ensure that the system is running as intended.
6. Maintenance:
   - Update the system as required.
   - Fix bugs and improve features.

A SIMPLE LAYOUT OF THE DIAGRAM

```
+-------------------+
| 1. Requirement    |
|    Analysis       |
+-------------------+
          |
          V
+-------------------+
| 2. Design         |
|    - Class Diagram|
|    - Architecture |
+-------------------+
          |
          V
+-------------------+
| 3. Implementation |
|    - Coding       |
|    - Encapsulation|
+-------------------+
          |
          V
+-------------------+
| 4. Testing        |
|    - Unit Tests   |
|    - Integration  |
+-------------------+
          |
          V
+-------------------+
| 5. Deployment     |
|    - Production   |
+-------------------+
          |
          V
+-------------------+
| 6. Maintenance    |
|    - Updates      |
|    - Bug Fixes    |
+-------------------
```

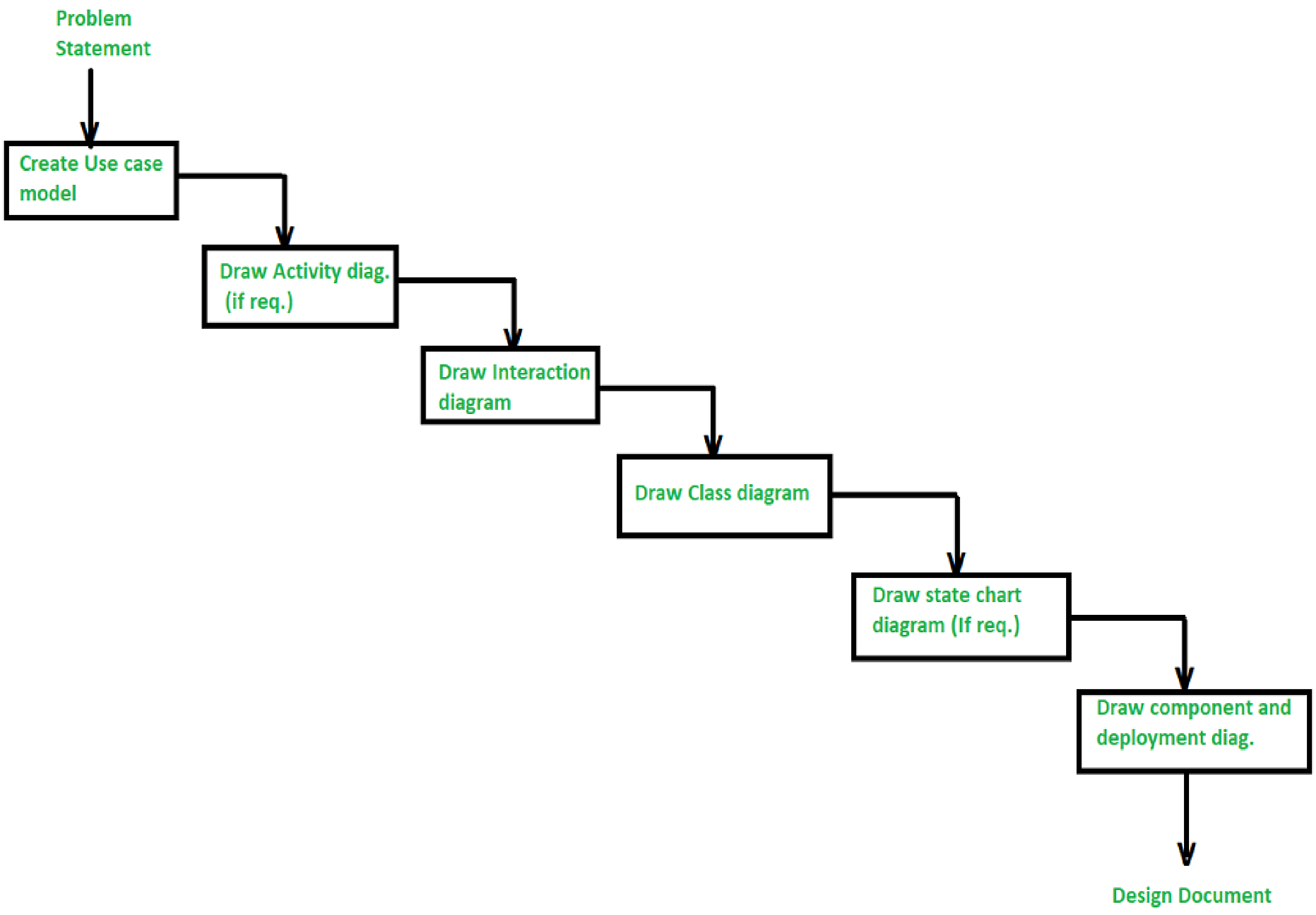
Explanation of Diagram:

- The Requirement Analysis step involves understanding what is needed from the system. This is where you gather all the necessary information and prepare for the design phase.

- In the Design phase, you create models for your system. This often includes creating class diagrams that show the classes, their attributes, their methods, and the relationships between them, such as inheritance and associations.

- Implementation is the coding phase where you write the actual code for the classes you have designed. Here, you apply the principle of encapsulation to protect the data within your objects, ensuring that only the methods provided are used to interact with the data.

- Testing is where you verify that your code works as intended. You might write unit tests for individual classes and integration tests to check how different parts of your system work together.


Problem Statement
Create Use case model
Draw Activity diag. (if req.)
Draw Interaction diagram
Draw Class diagram
Draw state chart diagram (If req.)
Draw component and deployment diag.
Design Document

- Deployment is the process of putting your system into production so that it can be used in a real-world environment.

- Finally, Maintenance is an ongoing process where the system is updated, bugs are fixed, and new features can be added. This step ensures that the system remains functional and relevant over time.

https://www.freecodecamp.org/news/java-object-oriented-programming-system-principles-oops-concepts-for-beginners/

https://www.geeksforgeeks.org/steps-to-analyze-and-design-object-oriented-system/

ii. WHAT IS THE OBJECT MODELING TECHNIQUES (OMT)?

Object Modelling Technique (OMT) is real world-based modelling approach for software modelling and designing. It was developed basically as a method to develop object-oriented systems and to support object-oriented programming. It describes the static structure of the system.

Object Modelling Technique is easy to draw and use. It is used in many applications like telecommunication, transportation, compilers etc. It is also used in many real-world problems. OMT is one of the most popular object-oriented development techniques used now-a-days. OMT was developed by *James Rumbaugh*.

https://www.geeksforgeeks.org/software-engineering-object-modeling-technique-omt/

iii. COMPARE OBJECT-ORIENTED ANALYSIS AND DESIGN (OOAD) AND OBJECT ANALYSIS AND DESIGN (OOP).

Object-Oriented Analysis and Design (OOAD) is a software engineering methodology that involves using object-oriented concepts to design and implement software systems. OOAD involves a number of techniques and practices, including object-oriented programming, design patterns, UML diagrams, and use cases.

As the name suggests, Object-Oriented Programming or OOPs refers to languages that use objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

https://www.geeksforgeeks.org/object-oriented-analysis-and-design/

https://www.geeksforgeeks.org/introduction-of-object-oriented-programming/

iv. Discuss Main goals of UML.

Unified Modelling Language (UML)

is a general purpose modelling language, The main aim of UML is to define a standard way to visualize the way a system has been designed? It is quite similar to blueprints used in other fields of engineering. UML is not a programming language, it is rather a visual language. We use UML diagrams to portray the behavior and structure of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis.

UML (Unified Modeling Language) is a general-purpose, graphical modeling language in the field of Software Engineering. UML is used to specify, visualize, construct, and document the artifacts (major elements) of the software system.

https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction/

https://www.javatpoint.com/uml

v. DESCRIBE THREE ADVANTAGES OF USING OBJECT ORIENTED TO DEVELOP AN INFORMATION SYSTEM.

OOP stands for Object-Oriented Programming. As you can guess from it's name it breaks the program on the basis of the objects in it. It mainly works on Class, Object, Polymorphism, Abstraction, Encapsulation and Inheritance. Its aim is to bind together the data and functions to operate on them.

Some of the well-known object-oriented languages are Objective C, Perl, Java, Python, Modula, Ada, Simula, C++, Smalltalk and some Common Lisp Object Standard.

THE ADVANTEGES INCLUDE;

- We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity,
- OOP language allows to break the program into the bit-sized problems that can be solved easily (one object at a time).
- The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.

https://www.geeksforgeeks.org/benefits-advantages-of-oop/

vi. **BRIEFLY EXPLAIN THE FOLLOWING TERMS AS USED IN OBJECT-ORIENTED PROGRAMMING. WRITE A SAMPLE JAVA CODE TO ILLUSTRATE THE IMPLEMENTATION OF EACH CONCEPT.**

a. **CONSTRUCTOR**
b. **OBJECT**
c. **DESTRUCTOR**
d. **POLYMORPHISM**
e. **CLASS**
f. **INHERITANCE**

a. Constructor: A constructor is a special method that is used to initialize objects of a class. It has the same name as the class and is called automatically when an object of the class is created. A constructor can take parameters or have no parameters. Here is an example of a constructor in Java:

```
public class Car {
    String make;
    String model;
    int year;

    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
}
```

b. Object: An object is an instance of a class. It has state (attributes) and behavior (methods). In Java, objects are created using the new keyword followed by a call to a constructor. Here is an example of creating an object in Java:

```
Car myCar = new Car("Toyota", "Camry", 2022);
```

c. Destructor: Java does not have a destructor. Instead, it has a garbage collector that automatically frees up memory when an object is no longer being used.

d. Polymorphism: Polymorphism is the ability of an object to take on many forms. In Java, polymorphism is achieved through method overriding and method overloading. Method overriding is when a subclass provides its own implementation of a method that is already defined in its superclass. Method overloading is when a class has multiple methods with the same name but different parameters. Here is an example of method overriding in Java:

```
public class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound");
    }
}


public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The dog barks");
    }
}
```

e. Class: A class is a blueprint for creating objects. It defines the attributes and methods that an object will have. In Java, a class is defined using the **class** keyword. Here is an example of a class in Java:

```
public class Car {
    String make;
    String model;
    int year;

    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    public void start() {
        System.out.println("The car starts");
    }
}
```

f. Inheritance: Inheritance is a mechanism in which one class acquires the properties and methods of another class. The class that is being inherited from is called the superclass or parent class, and the class that is inheriting is called the subclass or child class. In Java, inheritance is achieved using the **extends** keyword. Here is an example of inheritance in Java:

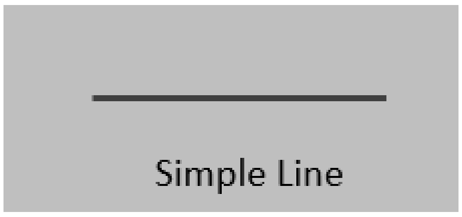
```
public class Animal {
    public void eat() {
        System.out.println("The animal eats");
    }
}


public class Dog extends Animal {
    public void bark() {
        System.out.println("The dog barks");
    }
}
```

vii. EXPLAIN THE THREE TYPES OF ASSOCIATIONS (RELATIONSHIPS) BETWEEN OBJECTS IN OBJECT ORIENTED.

Association

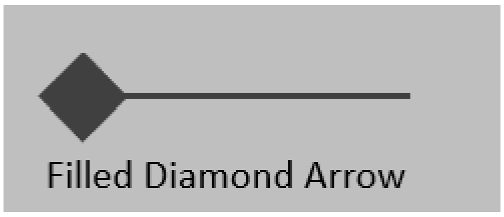
This relationship transpires when objects of one class know the objects of another class. The relationship can be one to one, one to many, many to one, or many to many. Moreover, objects can be created or deleted independently.



```
public static void main (String[] args) {
    Doctor doctorObj = new Doctor("Rick");
    Patient patientObj = new Patient("Morty");
    System.out.println(patientObj.getPatientName() +
        " is a patient of " + doctorObj.getDoctorName());
}
```

Composition

Is a "part-of" type of relationship, and is a strong type of association. In other words, composition happens when a class owns & contains objects of another class.

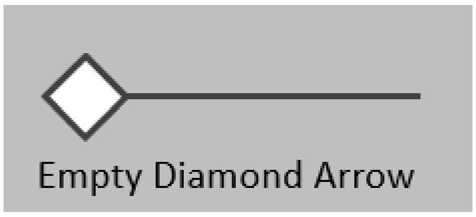


```
class Room {
//code here
}


class House {
   private Room;
//code here
}
```

Aggregation

Is a "has-a" type relationship and is a one-way form of association. This relationship exists when a class owns but shares a reference to objects of another class.



```
class Address{
//code here
}


class StudentClass{
   private Address studentAddress;
//code here
}
```

https://dev.to/tommyc/common-types-of-oop-relationships-and-their-uml-representation-5b27

viii. 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

The class diagram depicts a static view of an application. It represents the types of objects residing in the system and the relationships between them. A class consists of its objects, and also it may inherit from other classes. A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code.

It shows the attributes, classes, functions, and relationships to give an overview of the software system. It constitutes class names, attributes, and functions in a separate compartment that helps in software development. Since it is a collection of classes, interfaces, associations, collaborations, and constraints, it is termed as a structural diagram.

How to draw a Class Diagram?

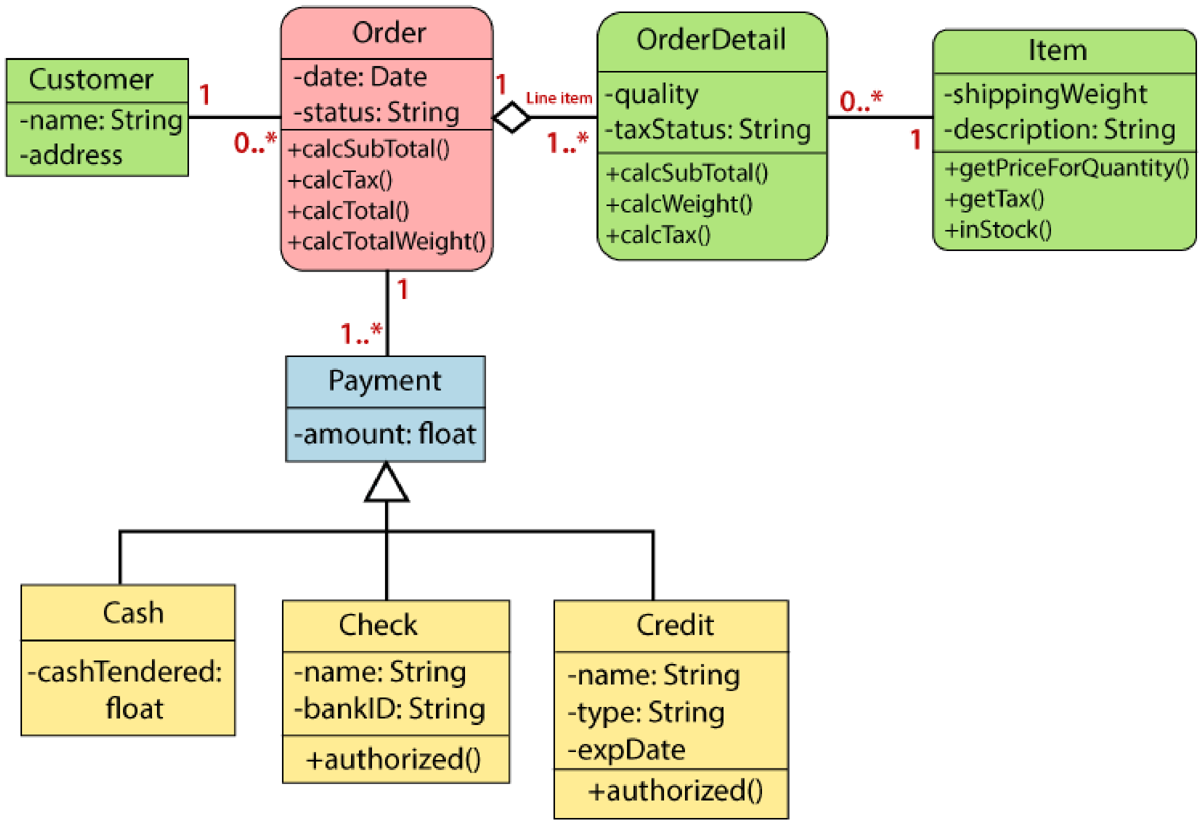
The class diagram is used most widely to construct software applications. It not only represents a static view of the system but also all the major aspects of an application. A collection of class diagrams as a whole represents a system.

Some key points that are needed to keep in mind while drawing a class diagram are given below:

1. To describe a complete aspect of the system, it is suggested to give a meaningful name to the class diagram.
2. The objects and their relationships should be acknowledged in advance.
3. The attributes and methods (responsibilities) of each class must be known.
4. A minimum number of desired properties should be specified as a greater number of the unwanted property will lead to a complex diagram.

5. Notes can be used as and when required by the developer to describe the aspects of a diagram.
6. The diagrams should be redrawn and reworked as many times to make it correct before producing its final version.

EXAMPLE INCLUDE;



ix. **GIVEN THAT YOU ARE CREATING AREA AND PERIMETER CALCULATOR USING C++, TO COMPUTER 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)

b. FRIEND FUNCTIONS

c. METHOD OVERLOADING AND METHOD OVERRIDING

d. LATE BINDING AND EARLY BINDING

e. ABSTRACT CLASS AND PURE FUNCTIONS

Area and Perimeter Calculator in C++ using OOP Concepts

This program demonstrates the use of various OOP concepts in a C++ area and perimeter calculator for circles, rectangles, triangles, and squares.

Note: Some implementations omit certain concepts for clarity and focus on the most relevant example. You can adapt and expand on these snippets to encompass all concepts fully.

a. Inheritance:

a.1) Single Inheritance**:**

```
class Shape { // Base class

public:

  virtual double area() = 0; // Pure virtual function (forces subclasses to implement area)

  virtual double perimeter() = 0; // Pure virtual function (forces subclasses to implement perimeter)
};


class Circle : public Shape { // Subclass inherits from Shape
public:
  Circle(double radius) : radius_(radius) {}
  double area() override { return M_PI * radius_ * radius_; }
  double perimeter() override { return 2 * M_PI * radius_; }
private:
  double radius_;
};
```

a.2) Multiple Inheritance:

```
class Polygon { // Base class for shapes with sides
public:
  virtual int numSides() = 0; // Pure virtual function (forces subclasses to
implement sides)
};


class Rectangle : public Shape, public Polygon {
public:
  Rectangle(double width, double height) : width_(width), height_(height) {}
  double area() override { return width_ * height_; }
  double perimeter() override { return 2 * (width_ + height_); }
  int numSides() override { return 4; }
private:
  double width_, height_;
};
```

a.3) Hierarchical Inheritance:

```
class Triangle : public Shape {
public:
  Triangle(double base, double height) : base_(base), height_(height) {}
  double area() override { return 0.5 * base_ * height_; }
  double perimeter() override { // ... implement based on triangle type }
private:
  double base_, height_;
};

class RightTriangle : public Triangle { // Inherits specific triangle type
public:
  RightTriangle(double base, double height) : Triangle(base, height) {}
  double perimeter() override { return base_ + height_ + hypot(base_, height_);
}
};
```

b. Friend Functions:

```
class Square : public Shape { // Access private side length for area calculation
public:
  Square(double side) : side_(side) {}
  friend double area(const Square& square) { // Friend function calculates area
    return square.side_ * square.side_; // Access private side_ directly
  }
  double perimeter() override { return 4 * side_; }
private:
  double side_;
};
```

c. Method Overloading and Overriding:

```
class Triangle {
public:
  double area(double base, double height) { return 0.5 * base * height; } // Overloaded for different input types
  double area() override { // Same function overrides inherited version in specific implementations
    // Area calculation based on triangle type (e.g., equilateral, isosceles)
  }
};
```

d. Late Binding and Early Binding:

Late Binding:

```
Shape* shape = new Circle(5); // Dynamically allocate object at runtime
double area = shape->area(); // Late binding - function call resolved at runtime based on actual object type


delete shape; // Release allocated memory
```

Early Binding:

```
Circle circle(5); // Statically allocate object at compile time
double area = circle.area(); // Early binding - function call resolved at compile time based on declared type
```

e. Abstract Class and Pure Functions:

```
abstract class Shape { // Abstract class with only pure virtual functions
public:
  virtual double area() = 0;
  virtual double perimeter() = 0;
};

// Subclasses (Circle, Rectangle, Triangle, etc.) implement pure virtual functions
for their specific shapes
```

X. USING A PROGRAM WRITTEN IN C++, DIFFERENTIATE BETWEEN THE FOLLOWING.

a) **FUNCTION OVERLOADING AND OPERATOR OVERLOADING**
b) **PASS BY VALUE AND PASS BY REFERENCE**
c) **PARAMETERS AND ARGUMENTS**

```
#include <iostream>

// Function overloading example
int add(int a, int b) { return a + b; }
double add(double a, double b) { return a + b; }

// Pass by value and reference example
void swap(int a, int b) {
  int temp = a;
  a = b;
  b = temp; // Changes local copies only, original values remain unchanged
}
```

```
void swapByRef(int& a, int& b) {
  int temp = a;
  a = b;
  b = temp; // Changes original values through references
}

// Parameters and arguments example
void printInfo(std::string name, int age = 18) { // "age" has a default value
  std::cout << "Name: " << name << ", Age: " << age << std::endl;
}

int main() {
  // Function overloading
  int sumInt = add(5, 3);
  double sumDouble = add(2.5, 1.2);
  std::cout << "sumInt: " << sumInt << std::endl;
  std::cout << "sumDouble: " << sumDouble << std::endl;

  // Pass by value and reference
  int x = 10, y = 20;
  swap(x, y); // Original values remain unchanged
  std::cout << "x after swap by value: " << x << ", y: " << y << std::endl;
  swapByRef(x, y); // Original values swapped
  std::cout << "x after swap by reference: " << x << ", y: " << y << std::endl;

  // Parameters and arguments
  printInfo("John"); // Uses default age
  printInfo("Jane", 30); // Overrides default age


  return 0;
}
```

Explanation:

1. Function overloading: As shown, two add functions exist with different parameter types (int and double), allowing you to perform addition depending on the data type.
2. Pass by value and reference: swap takes values by copy, modifying only local copies within the function, while swapByRef uses references to directly modify the original variables passed as arguments.
3. Parameters and arguments: The printInfo function defines parameters ("name" mandatory, "age" optional). When calling the function, you provide arguments that match the parameters.

XI. 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 +vit+xi $v(t)$=at+vi

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

} }
```

```
public class CalculateG {


    private double gravity = -9.81; // Earth's gravity in m/s^2
    private double fallingTime = 30; // Time of fall in seconds
    private double initialVelocity = 0.0; // Initial velocity (assumed zero)
    private double finalVelocity; // Final velocity after fall (calculated)
    private double initialPosition = 0.0; // Initial position (assumed zero)
    private double finalPosition; // Final position after fall (calculated)

    // Method for multiplication
    public double multiply(double a, double b) {
        return a * b;
    }

    // Method for squaring a number
    public double square(double a) {
        return multiply(a, a);
    }

    // Method for summation
    public double sum(double a, double b) {
        return a + b;
    }

    // Method for printing a result with label
    public void outline(String label, double result) {
        System.out.println(label + ": " + result + " m");
    }


    public static void main(String[] args) {
        CalculateG calculator = new CalculateG();


        // Calculate final velocity
        calculator.finalVelocity = sum(multiply(calculator.gravity,
calculator.fallingTime), calculator.initialVelocity);


        // Calculate final position
        calculator.finalPosition = sum(multiply(0.5, multiply(calculator.gravity,
square(calculator.fallingTime))),
                                    multiply(calculator.initialVelocity,
calculator.fallingTime),
                                    calculator.initialPosition);


        // Print final position
        calculator.outline("Final position after fall", calculator.finalPosition);


        // Print final velocity (modify outline method if needed)
        calculator.outline("Final velocity after fall", calculator.finalVelocity);
    }
}
```

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

```
class CalculateG {
public:
    CalculateG(double a, double b) : a_(a), b_(b) {}

    double multiply() { return a_ * b_; }

    double square() { return a_ * a_; }

    double sum() { return a_ + b_; }

    void print_result(double result) {
        std::cout << "The result is: " << result << std::endl;
    }

private:
    double a_;
    double b_;
};

int main() {
    // Example usage
    CalculateG g(5, 3);

    // Multiply
    double result = g.multiply();
    g.print_result(result);

    // Square
    result = g.square();
```

```
   g.print_result(result);


   // Sum
   result = g.sum();
   g.print_result(result);


   return 0;
}
```

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>


long fibonacci_even_sum(long long limit) {
  long a = 1, b = 2, even_sum = 0;
  while (b <= limit) {
    if (b % 2 == 0) {
      even_sum += b;
    }
    a = b;
    b = a + b;
  }
  return even_sum;
}


int main() {
  long limit = 4000000; // You can modify this limit
  long even_sum = fibonacci_even_sum(limit);
  std::cout << "The sum of all even-valued terms in the Fibonacci sequence up
to " << limit << " is: " << even_sum << std::endl;
  return 0;
}
```

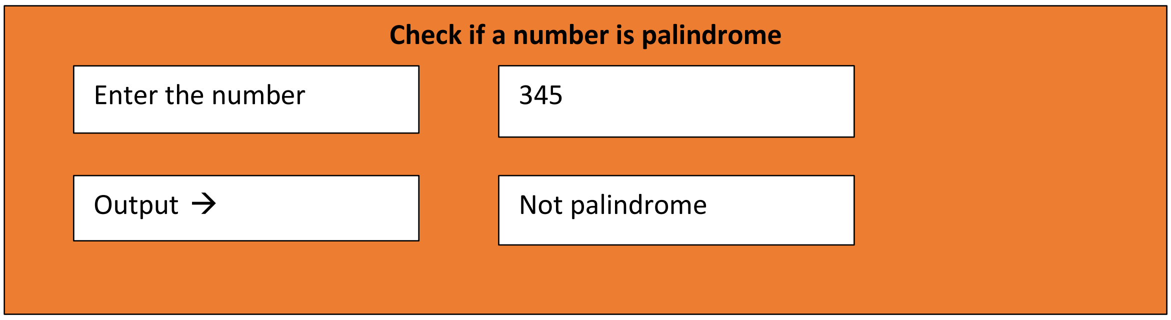
Explanation:

1. The function fibonacci_even_sum takes a single argument limit, which is the upper limit for the Fibonacci sequence.
2. We initialize three variables: a, b, and even_sum. a and b are used to store the previous two Fibonacci numbers, and even_sum is used to accumulate the sum of all even-valued terms.
3. We enter a loop that continues as long as the current Fibonacci number (b) is less than or equal to the limit.
4. Inside the loop, we check if the current number (b) is even. If it is, we add it to the even_sum.
5. Then, we update the values of a and b for the next iteration. a becomes the current value of b, and b becomes the sum of the previous two values (a and b).
6. After the loop, even_sum will contain the sum of all even-valued terms in the Fibonacci sequence up to the limit.

7. The main function sets the limit to 4 million (you can modify this) and calls the fibonacci_even_sum function to calculate the sum. It then prints the result to the console.

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:



```
#include <wx/wx.h>


class MyFrame : public wxFrame {
public:
  MyFrame(const wxString& title);
private:
  wxTextCtrl* numberInput;
  wxButton* checkButton;
  wxStaticText* resultLabel;
  void OnCheckButton(wxCommandEvent& event);
  DECLARE_EVENT_TABLE();
};

BEGIN_EVENT_TABLE(MyFrame, wxFrame)
  EVT_BUTTON(wxID_ANY, MyFrame::OnCheckButton)
END_EVENT_TABLE()

MyFrame::MyFrame(const wxString& title)
  : wxFrame(NULL, wxID_ANY, title, wxPoint(50, 50), wxSize(300, 150))
{
  numberInput = new wxTextCtrl(this, wxID_ANY, "", wxPoint(10, 10),
wxSize(200, 25));
  checkButton = new wxButton(this, wxID_ANY, "Check", wxPoint(10, 40));
  resultLabel = new wxStaticText(this, wxID_ANY, "", wxPoint(10, 70));

  Bind(wxEVT_BUTTON, &MyFrame::OnCheckButton, checkButton);
}

void MyFrame::OnCheckButton(wxCommandEvent& event) {
  long number;
  wxString input = numberInput->GetLineText(0);
  if (!wxAtoi(input, &number)) {
    resultLabel->SetLabel("Invalid Input!");
    return;
  }
  long reversedNumber = 0;
  long originalNumber = number;
  while (number > 0) {
    long digit = number % 10;
    reversedNumber = reversedNumber * 10 + digit;
    number /= 10;
  }

  if (originalNumber == reversedNumber) {
    resultLabel->SetLabel(wxString::Format("%ld is a palindrome!",
originalNumber));
  } else {
    resultLabel->SetLabel(wxString::Format("%ld is not a palindrome!",
originalNumber));
  }
}

wxGET_APP_FACTORY_CLASS(MyFrame)

int main(int argc, char **argv) {
  wxApp app;
  MyFrame* frame = new MyFrame("Palindrome Checker");
  frame->Show(true);
  app.MainLoop();
  return 0;
}
```

Explanation:

- The code defines a `MyFrame` class inheriting from `wxFrame` which is the base class for a window in wxWidgets.
- The `MyFrame` constructor creates and positions controls like `wxTextCtrl` for user input, `wxButton` to initiate checking, and `wxStaticText` to display the result.
- The `OnCheckButton` function is triggered when the Check button is clicked. It:
  - Reads the user input from the text control.
  - Converts the string input to a long integer using `wxAtoi`.
  - If the input is not a valid integer, an error message is displayed.
  - Otherwise, it implements a loop to:
    - Extract the digits of the original number one by one and add them to the reversed number in the appropriate order.
  - Compares the original and reversed numbers.
  - Based on the comparison, it displays either a palindrome confirmation or a non-palindrome message.
- The `main` function initializes the wxWidgets application and creates a `MyFrame` instance. It displays the frame and enters the main loop for user interaction.

Building and Running:

1. Save the code as `palindrome_checker.cpp`.
2. Download and install the wxWidgets library according to your platform.
3. Include the necessary header files and library paths during compilation. You can use a Makefile or IDE build options to simplify this process.
4. Compile and run the program.
5. Enter a number in the text box and click the Check button. The program will display the result as a palindrome or not.