ALLAMA IQBAL OPEN UNIVERSITY

USMAN NOTES
www.usmannotes.com

Name:

Registration No.:

Course Code: 3411

Program: BS

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Programming Language-II (3411) • Level: BS

Semester: Autumn 2026

Assignment No. 1

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE — For Study & Reference

Note: This guide is prepared for study and reference purposes only. Students are advised to read the relevant units of the course book, understand the concepts and the logic of every program, practise the programs on a computer, and then write the assignment in their own words. Copying this guide word for word is not allowed; the printed instructions on the assignment booklet also require answers in the student's own words. All Java programs in this guide are complete and self-contained: type them exactly, compile them with javac and run them to confirm the sample output before writing them in the assignment.

USMAN NOTES www.usmannotes.com

Question 1: Composition in Object-Oriented Programming (20 Marks)

Question as printed: Explain the concept of composition in object-oriented programming. Write a Java program showing a class that contains another class. (20)

Introduction

Object-oriented programming builds software out of classes and objects, and one of the most important questions in this style of programming is how classes are related to each other. Two

famous relationships are inheritance, which is an is-a relationship, and composition, which is a has-a relationship. Composition means that one class contains an object of another class as one of its own fields, so that the bigger object is literally built out of smaller objects, just as real-world things are built out of parts. A car has an engine, a human body has a heart, a university has departments and a computer has a processor and memory. In each case the whole is incomplete without its parts. This answer first explains the concept of composition in simple words, then shows how it is different from aggregation and inheritance, and finally gives a complete Java program in which a Car class contains an Engine object, together with its output and a step-by-step explanation, so that the idea becomes clear both in theory and in practice.

What Is Composition?

Composition is a design technique in which a class achieves its functionality by containing objects of other classes instead of inheriting that functionality from a parent class. The contained object is called the part, component or owned object, and the containing object is called the whole, owner or composite object. The defining feature of composition is strong ownership. The part is normally created inside the whole, usually inside the constructor of the whole, and the part cannot meaningfully exist independently of the whole. When the whole object is destroyed, its parts are also destroyed and become available for garbage collection. For example, if a Car object is created with the new operator, its Engine object is created along with it, and when the Car object is removed from memory, the Engine goes with it, because nobody else holds a separate, independent reference through which the Engine was supplied from outside.

In Java, composition is implemented in a very simple way: a field of a class is declared with the type of another class. If the class Car declares a field of the form private Engine engine, then every Car object contains a reference to an Engine object, and the methods of Car can call the methods of Engine through this reference, for example engine.start(). Nothing special is added to the language for composition; it is simply the disciplined use of object references as fields, combined with the rule that the whole creates and controls the lifetime of its parts. Because the part is usually declared private, outside code cannot reach it directly and must work through the public methods of the whole, which keeps the design encapsulated, safe and easy to understand.

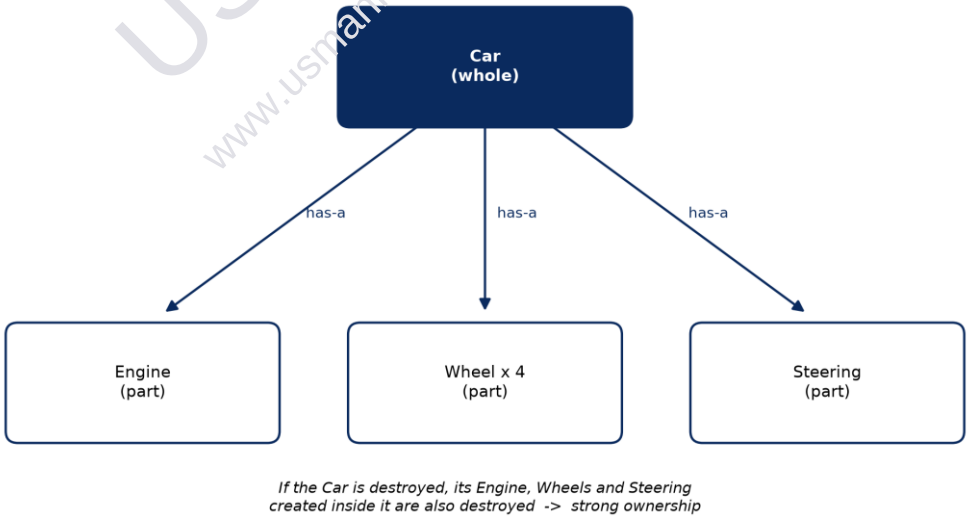


Figure 1: Composition — a Car (whole) is composed of an Engine, Wheels and a Steering (parts). The parts live and die with the whole.

Composition, Aggregation and Inheritance

Students often confuse composition with two neighbouring ideas, so the differences must be clear. Aggregation is also a has-a relationship, but it is a weak one: the part is created outside the whole and is only passed in, for example a Department receives already-existing Teacher objects. If the Department is closed, the Teachers still exist and can join another department, so the part survives the whole. Composition is the opposite: the part is born with the whole and dies with the whole, like a room in a house. Inheritance is different in kind, because it expresses an is-a relationship: a Dog is an Animal, so Dog can extend Animal and reuse its code through the class hierarchy. Inheritance couples a subclass tightly to its parent and can only be used sensibly when the is-a sentence sounds true. Experienced designers follow the well-known advice to favour composition over inheritance, because a composed object can change its parts at design level, parts can be replaced by better versions, and the design stays flexible instead of being locked into a rigid family tree.

Table 1: Composition compared with aggregation and inheritance

Feature	Composition	Aggregation	Inheritance
Relationship	Has-a (strong)	Has-a (weak)	Is-a
Lifetime of part	Part is created and destroyed with the whole	Part exists independently of the whole	Not applicable
How the part is obtained	Created inside the whole, usually in its constructor	Created outside and passed in as a parameter	Inherited from a parent class

Example	Car and its Engine; House and its Rooms	Department and its Teachers; Team and its Players	Dog extends Animal; Car extends Vehicle
Coupling	Whole controls its parts completely	Loose; parts can be shared	Very tight; subclass depends on parent

Advantages of Composition

Composition brings several practical advantages. First, it produces reusable parts: the same Engine class can be used inside a Car, a Truck or a Generator, without copying code. Second, it gives a natural model of the real world, because the structure of the program mirrors the structure of the problem, so the code is easy to read and explain. Third, it supports encapsulation: the whole can hide its parts and expose only a few simple methods, so changes inside a part do not disturb the rest of the program. Fourth, it is flexible: the behaviour of the whole can be changed by changing a part, for example by fitting a stronger engine, without rewriting the whole class. Finally, composition avoids the fragile-base-class problems of deep inheritance trees, in which a small change in a top-level parent class can break many subclasses at once. For all these reasons, composition is considered one of the central design tools of object-oriented programming.

The Java Program: A Car Class That Contains an Engine Class

The program below demonstrates composition in the clearest possible way. There are two part-and-whole classes besides the main class. The Engine class is the part: it has two fields, type and

horsePower, a constructor, and two small methods. The Car class is the whole: it declares a private field of type Engine, and, most importantly, it creates the Engine object inside its own constructor with the new operator. No Engine object is created in main and passed in, and that single detail is exactly what makes this composition rather than aggregation. The Car methods use the contained Engine by calling its methods through the engine reference. The third class, CompositionDemo, only creates a Car and asks it to display its details and start, which shows how the client code stays simple while the Car internally works through its part. Save the program in a file named CompositionDemo.java, because the public class is CompositionDemo.

Java Program — CompositionDemo.java

```
class Engine {
   private String type;
   private int horsePower;

   public Engine(String type, int horsePower) {
      this.type = type;
      this.horsePower = horsePower;
   }

   public void start() {
      System.out.println("Engine started: " + type
            + " engine with " + horsePower + " HP.");
   }

   public void showDetails() {
```

```
        System.out.println("Engine Type : " + type);
        System.out.println("Horse Power : " + horsePower + " HP");
    }
}

class Car {
    private String model;
    private Engine engine;   // composition: Car HAS-A Engine

    public Car(String model, String engineType, int horsePower) {
        this.model = model;
        // The part is created INSIDE the whole.
        // It will be destroyed when the Car is destroyed.
        this.engine = new Engine(engineType, horsePower);
    }

    public void startCar() {
        System.out.println("Starting the car " + model + " ...");
        engine.start();
    }

    public void showCar() {
        System.out.println("Car Model   : " + model);
        engine.showDetails();
    }
}
```

USMAN NOTES www.usmannotes.com

```
public class CompositionDemo {
   public static void main(String[] args) {
      Car car = new Car("Toyota Corolla", "Petrol", 130);
      car.showCar();
      car.startCar();
   }
}
```

Sample Output

```
Car Model    : Toyota Corolla
Engine Type : Petrol
Horse Power : 130 HP
Starting the car Toyota Corolla ...
Engine started: Petrol engine with 130 HP.
```

Explanation of the Program

The working of the program can be followed in a few steps. When main executes new Car(...), memory is allocated for a Car object and the Car constructor runs. Inside the constructor, the statement new Engine(engineType, horsePower) creates the Engine object and stores its reference in the private field engine, so the Car and its Engine come into existence together, which is the heart of composition. The call car.showCar() prints the model from the Car itself and then delegates the engine part of the work to engine.showDetails(); this delegation, where

the whole forwards a part of a task to its component, is the typical way composed objects cooperate. Similarly, car.startCar() does not know how an engine starts; it simply calls engine.start() and lets the part do its own work. If, later in a longer program, the reference car were set to null and the garbage collector removed the Car object, the Engine object would also become unreachable and would be removed, because no other part of the program holds it. The program therefore shows creation, use through delegation, and shared lifetime, the three signs of true composition.

Conclusion

To conclude, composition is the has-a relationship in which a class contains an object of another class as its own field, creates that part inside itself, uses it through delegation, and shares its lifetime with it: the part is born with the whole and dies with the whole. It differs from aggregation, where parts survive independently, and from inheritance, which models an is-a relationship and couples classes tightly. The Java program given above, in which the Car class creates and contains its Engine inside its constructor and drives it through simple method calls, is a complete and practical demonstration of the concept, and the same pattern can be applied to any real system, such as a Student composed of an Address, a Computer composed of a Processor and Memory, or an Invoice composed of its Items.

USMAN NOTES www.usmannotes.com

Question 2: Two-Dimensional Array — Creating and Displaying a Matrix (20 Marks)

Question as printed: Write a Java program to create a 2D array (matrix) of integers, take user input, and display the matrix in tabular form. (20)

Introduction

An array stores many values of the same type under one name, and a two-dimensional array goes one step further: it stores values in a grid of rows and columns, exactly like a table or a mathematical matrix. Two-dimensional arrays are needed whenever data is naturally tabular, for example the marks of several students in several subjects, the seats of a cinema hall, a chess board, an image made of pixels, or the distances between pairs of cities. Java supports such arrays directly, and in Java a two-dimensional array is technically an array of arrays: the outer array holds references to row arrays, and each row array holds the actual values. This answer explains the declaration, creation and indexing of a 2D array, and then presents a complete program that asks the user for the number of rows and columns, takes every element as input from the keyboard, and finally displays the complete matrix neatly in tabular form, followed by the sample output and a full explanation.

Declaring and Creating a 2D Array in Java

A two-dimensional array of integers is declared with two pairs of square brackets, for example int[][] matrix; and it is created by giving the number of rows and columns, for example matrix = new int[3][4]; which creates a grid of 3 rows and 4 columns, that is, 12 integer locations in total.

Declaration and creation can be combined in one statement. Every element is identified by two indexes: the row index and the column index, written as matrix[row][column]. Like all Java arrays, indexes start from zero, so the first element is matrix[0][0] and, in a 3 by 4 array, the last element is matrix[2][3]. When an int array is created, all elements are automatically initialised to zero, so a newly created matrix never contains garbage values. The number of rows is available as matrix.length, and the number of columns in any row is available as matrix[row].length, which is very useful when writing loops whose limits must adjust themselves to the size chosen by the user.

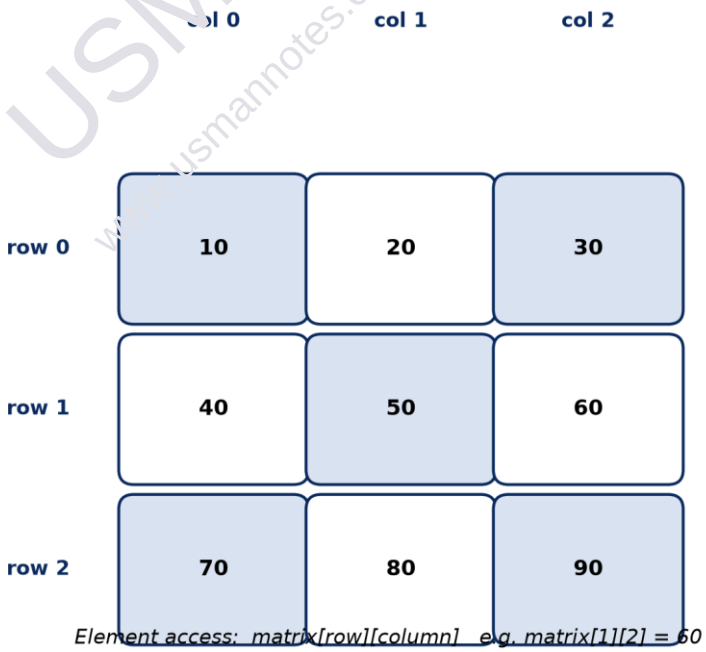


Figure 1: A 3 x 3 two-dimensional array. Every element is reached by its row and column index, for example matrix[1][2] = 60.

Taking Input and Displaying in Tabular Form

Two nested loops are the natural tool for handling a 2D array. The outer loop moves from row to row, and the inner loop moves across the columns of the current row. To take input, the program uses the Scanner class from the java.util package: a Scanner object connected to System.in reads integers typed by the user through its nextInt() method. For display, the program again uses nested loops, but this time it prints each element with System.out.printf in a fixed-width field, such as %5d, which reserves five character positions for every number. Fixed-width printing is what makes the output a proper table: every column lines up vertically under the column before it, whatever the size of the numbers. A simple print of values separated by spaces would look untidy as soon as numbers of different lengths, such as 7 and 450, appeared in the same column, so formatted output is an important part of this question.

The Java Program

The program below is general: it does not fix the size of the matrix in advance. It first asks the user how many rows and columns are required, creates an array of exactly that size, then prompts for each element by showing its position, such as [0][1], so the user always knows which cell is being filled. After the input is complete, it prints the matrix in tabular form using formatted output. The program should be saved as MatrixDemo.java, compiled with javac MatrixDemo.java and run with java MatrixDemo. Students should run it at least twice, once with a square matrix such as 3 by 3 and once with a rectangular matrix such as 2 by 4, to confirm that the loops and the display work correctly for every size.

Java Program — MatrixDemo.java

```
import java.util.Scanner;


public class MatrixDemo {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);


      System.out.print("Enter number of rows: ");
      int rows = input.nextInt();
      System.out.print("Enter number of columns: ");
      int cols = input.nextInt();


      // Create the 2D array (matrix) of integers
      int[][] matrix = new int[rows][cols];


      // Take user input for every element
      System.out.println("Enter the elements of the matrix:");
      for (int i = 0; i < rows; i++) {
         for (int j = 0; j < cols; j++) {
            System.out.print("Element [" + i + "][" + j + "]: ");
            matrix[i][j] = input.nextInt();
         }
      }


      // Display the matrix in tabular form
      System.out.println("The matrix in tabular form is:");
```

USMAN NOTES www.usmannotes.com

```
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.printf("%5d", matrix[i][j]);
            }
            System.out.println();  // move to the next row
        }

        input.close();
    }
}
```

Sample Output

```
Enter number of rows: 3
Enter number of columns: 3
Enter the elements of the matrix:
Element [0][0]: 10
Element [0][1]: 20
Element [0][2]: 30
Element [1][0]: 40
Element [1][1]: 50
Element [1][2]: 60
Element [2][0]: 70
Element [2][1]: 80
Element [2][2]: 90
The matrix in tabular form is:
   10   20   30
```

USMAN NOTES www.usmannotes.com

```
40   50   60
70   80   90
```

Explanation of the Program

The explanation follows the flow of execution. The Scanner object input is created first so that all keyboard reading goes through one object. The values read into rows and cols decide the size of the array in the statement new int[rows][cols]; this is possible because Java creates arrays at run time, so their size may come from the user instead of being written in the code. In the input loops, the outer variable i selects the row and the inner variable j selects the column, and the assignment matrix[i][j] = input.nextInt() stores each typed number in exactly the cell named by the prompt. In the display loops the same pair of indexes visits the cells in the same order, printf with %5d prints each value right-aligned in a field of width five, and println() after the inner loop finishes a row and starts a new line, which is what gives the output its table shape. Finally, input.close() releases the Scanner resource, which is good practice in every program that opens a stream.

Table 1: Important points about 2D arrays used in this program

Point	Detail
Declaration	Two pairs of brackets: int[][] matrix;
Creation	new int[rows][cols] creates rows x cols integer locations, all zero at first
Accessing an element	matrix[row][column], with both indexes starting from 0

Number of rows	matrix.length
Number of columns in a row	matrix[i].length
Traversal pattern	Nested loops: outer loop for rows, inner loop for columns
Tabular display	Formatted printing such as printf("%5d", value) keeps columns aligned

Common Mistakes to Avoid

Three mistakes are common in this program and should be avoided. The first is an index error: writing matrix[rows][cols] instead of matrix[rows-1][cols-1] for the last element throws an ArrayIndexOutOfBoundsException, because valid indexes run from zero to length minus one. The second is forgetting the line break after each row in the display loop, which prints the whole matrix on one long line instead of in tabular form. The third is reading input in one order and displaying in another without realising that the first index is always the row and the second is always the column; swapping them transposes the matrix. Keeping the simple rule row first, column second in both input and output loops avoids all three problems, and testing the program with unequal rows and columns, such as 2 rows and 4 columns, is the quickest way to catch a swapped index, because a square test matrix can hide the mistake.

Conclusion

To conclude, a two-dimensional array in Java is an array of row arrays, declared with two pairs of brackets, created with the new operator, and accessed through a pair of zero-based indexes

in the form matrix[row][column]. The program given above shows the complete standard pattern for working with such an array: it takes the size from the user, fills every element through nested loops with the help of the Scanner class, and displays the finished matrix in neat tabular form using formatted output. Once this pattern is understood, it can easily be extended to useful operations such as adding two matrices, finding the sum of each row, or transposing a matrix, all of which use exactly the same nested-loop structure.

Question 3: The this Keyword – Local and Instance Variables (20 Marks)

Question as printed: Describe the concept of this keyword in Java. Write a Java program to differentiate between local and instance variables using this. (20)

Introduction

Every Java object carries its own data in its instance variables, and every method works with a mixture of those instance variables and its own local variables and parameters. When a local variable or a parameter happens to have the same name as an instance variable, the two become difficult to distinguish, and Java needs a clear way to say this name means the variable belonging to the current object. That way is the keyword this. The word this is one of the most useful small keywords in Java: inside any instance method or constructor, this is a reference to the very object on which the method was called. This answer explains the meaning of this, lists its main uses, explains the special problem of name shadowing that makes this necessary, and then gives a complete program in which a method receives a local parameter with exactly the

same name as an instance variable, so that the difference between the local variable and the instance variable, and the role of this in separating them, can be seen in the output itself.

What Is this in Java?

In Java, this is a reference variable that refers to the current object, meaning the object whose method or constructor is being executed at that moment. When an object calls a method, the Java Virtual Machine passes that object silently to the method, and inside the method the name this stands for it. Writing this.marks therefore means the marks field of the current object, while writing only marks may mean a local variable or parameter of the same name, if one exists. The keyword this cannot be used inside a static method, because a static method belongs to the class as a whole and is not called on any particular object, so there is no current object for this to refer to. It is also important that this is a final kind of reference in practice: a program cannot assign a new value to this itself, it can only use it to reach the fields and methods of the current object.

Instance Variables and Local Variables

An instance variable is declared inside the class but outside every method, and each object of the class receives its own separate copy of it. It lives as long as the object lives, it is stored on the heap with the object, and if the programmer does not initialise it, Java gives it a default value, such as 0 for int, 0.0 for double and null for references. A local variable is declared inside a method, constructor or block, including the parameters of a method. It belongs to that one execution of the method only, it is stored on the stack, it comes into existence when the method is called and disappears when the method ends, and Java gives it no default value, so it

must be assigned before use. When a method declares a local variable or parameter with the same name as an instance variable, the local name hides or shadows the instance variable inside that method: the simple name then refers to the local variable, and the instance variable can still be reached, but only through this.

Table 1: Instance variables and local variables compared

Feature	Instance Variable	Local Variable
Declared	Inside the class, outside all methods	Inside a method, constructor or block (parameters included)
Copies	One separate copy for every object	One copy for every call of the method
Lifetime	As long as the object exists	Only while the method is executing
Default value	Yes: 0, 0.0, false or null according to type	No default value; must be assigned before use
Access when names clash	Reached through this.variableName	Reached by the simple name
Storage	On the heap, as part of the object	On the stack, as part of the method call

Main Uses of this

The keyword this has several standard uses, and a good answer should know all of them. Its first and most common use is the one demonstrated in the program below: to refer to the

instance variables of the current object when a parameter or local variable has the same name, which is why constructors are so often written in the form this.name = name, where the left side is the field and the right side is the parameter. Its second use is to call another constructor of the same class, through the statement this(...), which must be the first statement of a constructor and which avoids repeating initialisation code. Its third use is to call a method of the current object explicitly, as in this.display(), although the this is optional there. Its fourth use is to pass the current object as an argument to another method or constructor, for example registering an object with a list. Its fifth use is to return the current object from a method, return this, which allows calls to be chained together. Of these, the first use, separating instance variables from local variables, is the one asked in this question and the one used most often in real programs.

USMAN NOTES www.usmannotes.com

The Java Program

In the program below, the class Student (which is also the public class, so the file must be named Student.java) has two instance variables, name and marks. The constructor receives parameters with exactly the same names, so it must use this.name and this.marks to store the values into the fields; without this, the constructor would assign the parameters to themselves and the fields would remain at their default values. The method compareMarks then receives one more parameter also called marks. Inside that method, the simple name marks prints the local parameter value, while this.marks prints the value stored in the object. Running the program with two different numbers makes the difference visible on the screen in two clearly labelled lines, which is exactly the differentiation the question asks for.

Java Program — Student.java

```
public class Student {
   // Instance variables: every Student object has its own copies
   private String name;
   private int marks;

   // Constructor: parameter names are the SAME as field names
   public Student(String name, int marks) {
      this.name = name;    // this.name  = instance variable
                  // name       = parameter (local)
      this.marks = marks;  // this.marks = instance variable
   }

   // A local parameter named marks shadows the instance variable
   public void compareMarks(int marks) {
      System.out.println("Local variable (parameter) marks    = " + marks);
      System.out.println("Instance variable (this.marks)      = " +
      this.marks);
   }

   public void display() {
      System.out.println("Student Name : " + this.name);
      System.out.println("Student Marks: " + this.marks);
   }
```

```
    public static void main(String[] args) {
        Student s1 = new Student("Ali Raza", 85);
        s1.display();
        System.out.println("--- Calling compareMarks(60) ---");
        s1.compareMarks(60);
    }
}
```

Sample Output

```
Student Name : Ali Raza
Student Marks: 85
--- Calling compareMarks(60) ---
Local variable (parameter) marks    = 60
Instance variable (this.marks)      = 85
```

Explanation of the Program

The output proves the distinction beyond doubt. The object s1 was created with marks equal to 85, and that value is stored in the instance variable, so display() and this.marks both report 85. When compareMarks(60) is called, the number 60 exists only as the local parameter of that method call. Inside the method the first line prints the plain name marks and Java understands the nearest declaration, which is the parameter, so it prints 60. The second line writes this.marks, which instructs Java to leave the local names alone and go to the field of the current object, so it prints 85. If the programmer removed this in the constructor and wrote only marks

= marks, the program would still compile, but it would assign the parameter to itself, the field would stay 0, and the student would wrongly appear to have zero marks; this classic bug is the practical reason the this keyword exists. Note also that main is static, so it never uses this; it works through the object reference s1 instead, which plays the same role from outside the object.

Conclusion

To conclude, this is a built-in reference to the current object, available inside every instance method and constructor, and its most important job is to distinguish the instance variables of that object from local variables and parameters that shadow them with the same name. An instance variable belongs to the object, lives with it and has a default value, while a local variable belongs to one method call, is stored on the stack and must be given a value before use. Writing this.marks reaches the field of the object, while writing marks reaches the local name, as the program above shows with the values 85 and 60 printed side by side. The same keyword also supports constructor chaining with this(...), explicit method calls, passing the current object and returning the current object, which makes it a small keyword with a central place in everyday Java programming.

Question 4: Method Overloading (20 Marks)

Question as printed: What is method overloading? Write a Java program to demonstrate method overloading using different parameter lists. (20)

Introduction

In everyday language we happily use one word for several closely related actions, and Java allows a class to do the same with its method names. Method overloading is the feature by which two or more methods in the same class share one name but have different parameter lists, so that the programmer can call, for example, add for two integers, add for three integers and add for two decimal numbers, without inventing a new name for every variation. Overloading is one of the two main forms of polymorphism in Java, the compile-time or static form, because the compiler decides which version of the method a call refers to by looking at the arguments supplied. This answer defines method overloading precisely, states the exact rules that make two methods overloaded rather than duplicated or overridden, explains how the compiler resolves an overloaded call, and then demonstrates everything with a complete calculator program that overloads one method name in four different ways, followed by output and explanation.

What Is Method Overloading?

Method overloading means defining more than one method with the same name in the same class, provided that their parameter lists are different. The parameter list can differ in three ways: in the number of parameters, in the types of the parameters, or in the order of the types of the parameters. For example, add(int, int), add(int, int, int), add(double, double) and add(String, String) are four different overloaded forms of one method named add, because no two of them have the same list of parameter types in the same order. The return type may be the same or different in overloaded methods, but the return type alone can never create

overloading: two methods that differ only in return type, such as int add(int, int) and double add(int, int), are illegal, because a caller who ignored the returned value would leave the compiler with no way to choose between them. Overloaded methods usually perform the same general task on different kinds of data, which is why sharing one meaningful name makes the class easier to learn and use.

Rules and How the Compiler Chooses

Three rules govern overloading. First, the methods must be in the same class (methods inherited from a parent can also be overloaded by a subclass, but the basic case is one class). Second, the names must be identical and the parameter lists must differ in number, type or order. Third, the choice between the versions is made at compile time: the compiler compares the types of the actual arguments in the call with the parameter types of every candidate method and selects the exact or best match, a process called overload resolution. If no candidate matches, the program does not compile, and if two candidates match equally well the compiler reports an ambiguous call. This compile-time decision is why overloading is called static or early binding. Method overloading must not be confused with method overriding: overriding occurs between a parent class and a subclass, keeps the name and the complete parameter list exactly the same, and is decided at run time, whereas overloading occurs inside one class, requires different parameter lists, and is decided at compile time.

Table 1: Method overloading and method overriding compared

Feature	Method Overloading	Method Overriding
Where it happens	Within the same class	Between a parent class and its

		subclass
Parameter list	Must be different (number, type or order)	Must be exactly the same
Return type	May differ, but cannot be the only difference	Must be the same or a compatible subtype
When the method is chosen	At compile time (static binding)	At run time (dynamic binding)
Also called	Compile-time polymorphism	Run-time polymorphism

The Java Program

The program below defines a class Calculator containing four overloaded versions of a method named add. The first version adds two int values, the second adds three int values, so it differs in the number of parameters; the third adds two double values, so it differs in the types of the parameters; and the fourth joins two String values, showing that overloading is not limited to numbers. The main method of the public class OverloadingDemo calls all four versions one after another. Because each call passes arguments of a different pattern, the compiler can match every call to exactly one version, and the output lines are labelled so that the reader can see which version produced which result. The program should be saved as OverloadingDemo.java.

Java Program — OverloadingDemo.java

```
class Calculator {


    // Version 1: two int parameters
```

```
    int add(int a, int b) {
        return a + b;
    }

    // Version 2: THREE int parameters (different number)
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Version 3: two double parameters (different types)
    double add(double a, double b) {
        return a + b;
    }

    // Version 4: two String parameters (different types)
    String add(String a, String b) {
        return a + b;
    }
}

public class OverloadingDemo {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        System.out.println("add(10, 20)          = " + calc.add(10, 20));
```

USMAN NOTES www.usmannotes.com

```
        System.out.println("add(10, 20, 30)      = " + calc.add(10, 20, 30));
        System.out.println("add(5.5, 2.5)        = " + calc.add(5.5, 2.5));
        System.out.println("add(Hello,  World)   = "
              + calc.add("Hello ", "World"));
    }
}
```

Sample Output

```
add(10, 20)          = 30
add(10, 20, 30)      = 60
add(5.5, 2.5)        = 8.0
add(Hello,  World)   = Hello World
```

Explanation of the Program

Each line of the output is produced by a different method, and it is instructive to trace how the compiler matched them. The call calc.add(10, 20) supplies two int arguments, so it is bound to version 1 and returns 30. The call calc.add(10, 20, 30) supplies three int arguments; version 1 cannot accept three values, so the call is bound to version 2 and returns 60. The call calc.add(5.5, 2.5) supplies two double arguments, and since a decimal literal is a double in Java, version 3 is selected and the result is the double value 8.0, printed with its decimal point, which itself proves that a different method ran. The last call passes two strings, so version 4 runs and joins them into Hello World. Nothing in the running program makes these decisions; all four bindings were fixed during compilation, which illustrates compile-time polymorphism in the

simplest possible form. Java itself uses overloading heavily in its own library: the println method of System.out, for example, is overloaded for int, double, String, boolean and every other type, which is why one name can print any kind of value.

Advantages of Method Overloading

Overloading improves a program in several ways. It increases readability, because related operations carry one natural name instead of a family of artificial names such as addInt, addThreeInt and addDouble. It makes a class easier to use, because the caller simply supplies the data available and lets the compiler find the matching version. It supports flexibility, since new versions can be added for new data types without changing the existing calls. It is also the standard way to provide several constructors for a class, because constructors are overloaded in exactly the same way: a class may offer a no-argument constructor, a constructor with one argument and a constructor with all arguments, and the caller chooses by the arguments supplied at creation time. Used sensibly, where every version genuinely performs the same kind of task, overloading keeps code clean; used carelessly, where one name does unrelated jobs, it confuses the reader, so the versions should always share one clear purpose.

Conclusion

To conclude, method overloading is the definition of several methods with the same name in the same class but with different parameter lists, differing in the number, the types or the order of parameters, while a difference in return type alone is not allowed. The compiler resolves every call at compile time by matching the arguments to the best-fitting version, which makes overloading the compile-time form of polymorphism, distinct from overriding, which

keeps the signature the same across a parent and a subclass and is resolved at run time. The Calculator program above demonstrates the feature completely, with one name, add, serving two integers, three integers, two doubles and two strings, and producing the correct different result for every call.

Question 5: Recursion — Factorial of a Number (20 Marks)

Question as printed: Explain the concept of recursion. Write a Java program to calculate the factorial of a number using a recursive method. (20)

Introduction

Most methods solve a problem by calling other, different methods, but some problems have a beautiful special structure: the problem itself contains a smaller copy of the same problem inside it. The factorial of a number is the classic example, because 5 factorial is 5 times 4 factorial, 4 factorial is 4 times 3 factorial, and so on until 1 factorial, which is simply 1. Recursion is the programming technique that matches this structure directly: a recursive method is a method that calls itself, each time with a smaller input, until the input becomes so small that the answer is known immediately. Recursion is used in searching and sorting algorithms, in processing trees and folders, in mathematical functions and in many elegant solutions where a loop would be clumsy. This answer explains the concept of recursion, describes its two compulsory parts, the base case and the recursive case, traces the factorial computation step by step, and presents a complete Java program that reads a number from the user and calculates its factorial with a recursive method.

What Is Recursion?

Recursion is a problem-solving technique in which a method solves a problem by calling itself to solve a smaller sub-problem of the same kind, and then combining the returned result with its own contribution. Every correct recursive method must contain two parts. The first part is the base case, also called the terminating case: the smallest form of the problem, which the method answers directly without calling itself again. The second part is the recursive case: the method reduces the problem, calls itself with the reduced input, and uses the value that comes back. If either part is missing or wrong, the method fails: without a base case, or with a recursive call that does not move towards the base case, the method calls itself forever, filling the call stack until Java stops the program with a StackOverflowError. Behind the scenes, every call, finished or unfinished, is stored as a separate frame on the call stack, holding its own copy of the parameters and local variables; this is why the waiting calls do not mix up their values, and it is also why very deep recursion consumes memory.

Factorial as a Recursive Problem

The factorial of a non-negative integer n, written n!, is the product of all whole numbers from 1 up to n, so 5! = 5 x 4 x 3 x 2 x 1 = 120, and by definition 0! = 1 and 1! = 1. This definition converts into a recursive definition almost word for word. The base cases are factorial(0) = 1 and factorial(1) = 1, which need no further calculation. The recursive case is factorial(n) = n x factorial(n - 1) for n greater than 1, which says exactly that a bigger factorial is built from the next smaller factorial. The process has two phases. In the winding phase, the calls pile up: factorial(5) waits for factorial(4), which waits for factorial(3), and so on down to factorial(1). In

the unwinding phase, the base case returns 1, each waiting call multiplies that result by its own n and returns upwards — 2, 6, 24 and finally 120 — until the original call receives the complete answer. Understanding these two phases is the key to understanding every recursive method, because the same winding and unwinding pattern appears in all of them.

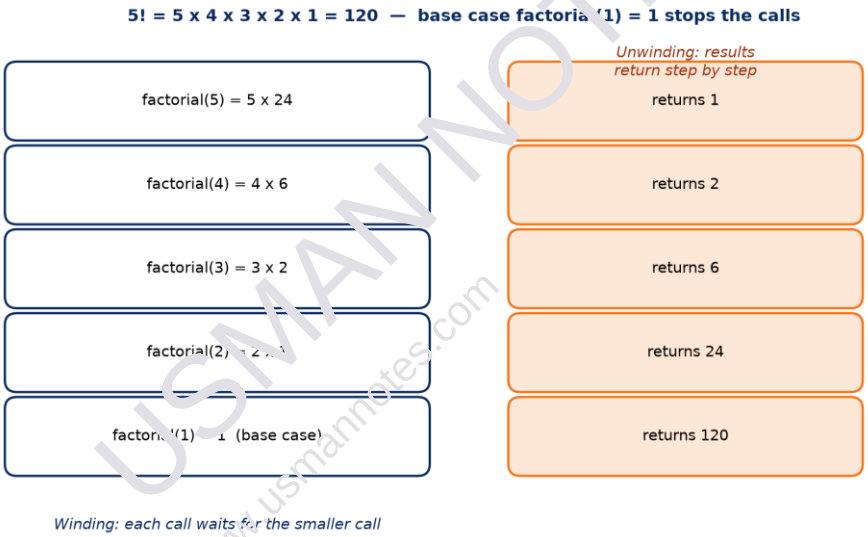


Figure 1: Recursion in factorial(5). Calls wind down to the base case factorial(1) = 1, then results unwind upwards: 1, 2, 6, 24, 120.

The Java Program

The program below contains a recursive method factorial that takes an int and returns a long. The return type is long rather than int because factorials grow very quickly — 13! already exceeds the largest int value — so long gives the program a safe range up to 20!. Inside the method, the if statement tests the base case: if n is 0 or 1, the method returns 1 immediately and makes no further call. Otherwise, the method returns n multiplied by factorial(n - 1), which is the recursive call with a smaller argument, exactly as the mathematics requires. The main

method reads the number from the user with a Scanner, rejects a negative number with a clear message because factorial is not defined for negative integers, and otherwise prints the result. The file must be saved as FactorialDemo.java, matching the public class name.

Java Program — FactorialDemo.java

```
import java.util.Scanner;


public class FactorialDemo {


   // Recursive method to calculate factorial
   static long factorial(int n) {
      if (n == 0 || n == 1) {
         return 1;            // base case: stops recursion
      } else {
         return n * factorial(n - 1);  // recursive case
      }
   }


   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);


      System.out.print("Enter a non-negative number: ");
      int number = input.nextInt();


      if (number < 0) {
```

```
            System.out.println("Factorial is not defined for negative numbers.");
        } else {
            long result = factorial(number);
            System.out.println("Factorial of " + number + " is " + result);
        }

        input.close();
    }
}
```

Sample Output

```
Enter a non-negative number: 5
Factorial of 5 is 120
```

Step-by-Step Trace of factorial(5)

Tracing the program for the input 5 shows recursion in action. The call factorial(5) finds that n is not 0 or 1, so it begins computing 5 x factorial(4) and waits. The call factorial(4) similarly waits on 4 x factorial(3), factorial(3) waits on 3 x factorial(2), and factorial(2) waits on 2 x factorial(1). At this point five frames are sitting on the call stack, each remembering its own value of n. Then factorial(1) meets the base case and returns 1 at once, without any new call. The waiting calls now finish one by one in reverse order: factorial(2) returns 2 x 1 = 2, factorial(3) returns 3 x 2 = 6, factorial(4) returns 4 x 6 = 24, and factorial(5) returns 5 x 24 = 120, which main prints. The student should notice two things in this trace: the argument decreases by exactly one in every

call, which guarantees that the base case is always reached, and no loop variable and no separate storage were needed, because the call stack itself remembered all the unfinished multiplications.

Table 1: Recursion compared with iteration (loops)

Feature	Recursion	Iteration (Loop)
Basic idea	A method calls itself with a smaller input	A block of code repeats with loop control variables
Stopping condition	Base case	Loop condition becomes false
Memory use	Higher: one stack frame for every call	Lower: usually one set of variables
Speed	Slightly slower because of call overhead	Usually faster
Best suited for	Problems that are naturally self-similar: factorial, tree traversal, binary search, Tower of Hanoi	Simple repetition: counting, printing, summing a list
Risk if written wrongly	StackOverflowError from endless calls	Infinite loop

Advantages, Limitations and Cautions

Recursion has real advantages: it gives short, clear code for self-similar problems, it mirrors the mathematical definition of functions such as factorial and Fibonacci numbers, and it is the natural and often the only simple method for structures such as trees, where a loop cannot

easily follow branches. It also has limitations that a careful programmer respects. Every call costs time and stack memory, so for a very large input a loop is more efficient and safer. A missing or unreachable base case is a fatal error, and an input that grows instead of shrinking between calls has the same effect. In the factorial program specifically, the input must also be kept moderate: even with a long result, 21! overflows, so the program is suitable for the small demonstration numbers used in a course assignment, and its negative-number check shows the habit of validating input before computing. The rule is to choose recursion where it makes the logic clearer — as it does here — and iteration where raw repetition is all that is needed.

Conclusion

To conclude, recursion is the technique in which a method calls itself with a smaller input until a base case stops the process and the partial results combine during unwinding. A correct recursive method always needs a base case that is actually reached and a recursive case that moves steadily towards it, with the call stack holding the waiting calls. Factorial is the ideal first example, because its mathematical definition, factorial(n) = n x factorial(n - 1) with factorial(0) = factorial(1) = 1, translates line by line into the Java method shown above, whose trace for 5 produces the correct result 120. Mastering this one example prepares the student for the many advanced topics, from binary search to tree processing, that are built on the same idea.