MODULE 2

1. What are constructors? Explain two types of constructors with an example program.

A constructor is a **special method** in a class that is automatically called when an object of the class is created. It is **used to initialize the newly created object.** Constructors share the same name as the class and do not have a return type.

Key Points:

- The constructor is invoked when an object is instantiated.
- It has the same name as the class.
- It does not have a return type (not even void).
- It is used to initialize instance variables of the class.

Types of Constructors:

1. Default Constructor
2. Parameterized Constructor

1. Default Constructor

A default constructor is a constructor that does not take any arguments. If no constructor is defined in a class, Java automatically provides a default constructor. It initializes the object with default values (like 0 for integers, null for objects).

Features:

- It does not accept parameters.
- Initializes objects with default values.

Syntax:

```
class ClassName {
    // Constructor
    public ClassName() {
        // Initialization code
    }
}
```

Example

```
class Car {
    String brand;
    int year;

    // Default constructor
    public Car() {
        brand = "Toyota"; // Default value
        year = 2020;      // Default value
    }

    public void display() {
        System.out.println(brand + " - " + year);
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Car car1 = new Car(); // Calls default constructor
        car1.display();       // Prints default values
    }
}
```

/// op Toyota - 2020

2. Parameterized Constructor

A parameterized constructor is a constructor that **accepts parameters.** This allows the user to initialize an object with specific values at the time of creation.

Features:

- It takes one or more parameters.
- Initializes an object with specific values.

Syntax:

```
class ClassName {
    // Constructor with parameters
    public ClassName(type parameter1, type parameter2) {
        // Initialization code
    }
}
```

Example :-

```
class Car {
    String brand;
    int year;

    // Parameterized constructor
    public Car(String b, int y) {
        brand = b; // Assign the value of b to brand
        year = y;  // Assign the value of y to year
    }

    public void display() {
        System.out.println(brand + " - " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Honda", 2022); // Calls parameterized constructor
        car1.display(); // Prints specific values
    }
}
```

// o/p Honda - 2022

2 Define recursion. Write a recursive program to find nth Fibonacci number.

Recursion is a process in which a function calls itself in order to solve a problem. It is typically used to break a large problem into smaller, manageable sub-problems. A recursive function must have:

1. **Base case:** A condition to stop the recursion.
2. **Recursive case:** The function calls itself.

Example: Recursive Program to Find nth Fibonacci Number

The Fibonacci series is a sequence where each number is the sum of the two preceding ones, starting from 0 and 1.

The Fibonacci sequence looks like this:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

Recursive Formula:

- **Base case:**
- **Fib(0) = 0**
- **Fib(1) = 1**
- **Recursive case: Fib(n) = Fib(n-1) + Fib(n-2)**

Simplified Example Program:

```
class Fibonacci {

    // Recursive method to find nth Fibonacci number
    public static int fibonacci(int n) {
        if (n == 0) {
            return 0; // Base case 1
        } else if (n == 1) {
            return 1; // Base case 2
        } else {
            return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
        }
    }

    public static void main(String[] args) {
        int n = 5;  // Find the 5th Fibonacci number
        System.out.println("Fibonacci of " + n + " is: " + fibonacci(n));
    }
}
```

Fibonacci of 5 is: 5

3. Explain the various access specifiers in Java.

Access specifiers in Java are keywords used to **define the visibility or scope of classes, methods, and variables.** They determine which parts of the program can access the particular members (variables, methods, etc.) of a class.

There are 4 types of access specifiers in Java:

1. Public
2. Private
3. Protected
4. Default (Package-Private)

1. Public Access Specifier

- **Visibility:** The member is accessible from anywhere (inside the same class, different classes, different packages).
- **Used for:** Public methods or variables that you want to be accessible from any class.

Example:

```
class MyClass {
    public int number = 10;

    public void display() {
        System.out.println("Number: " + number);
    }
}
public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        System.out.println(obj.number);  // Accessible
        obj.display();  // Accessible
    }
}
```

2. Private Access Specifier

- **Visibility:** The member is accessible only within the same class. Other classes cannot access private members.
- **Used for:** Protecting sensitive data inside a class (e.g., variables or methods).

Example:

```
class MyClass {
    private int number = 10;  // Private variable

    private void display() {  // Private method
        System.out.println("Number: " + number);
    }
```

```
    public void accessPrivateMethod() {
        display();  // Access private method from within the class
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        // obj.number = 10; // Error: number has private access
        // obj.display();  // Error: display() has private access
        obj.accessPrivateMethod();  // Allowed: Accessing via public method
    }
}
```

3. Protected Access Specifier

- **Visibility:** The member is accessible within the same package or by subclasses (even if they are in different packages).
- **Used for:** Allowing access to subclasses while restricting access to other classes.

Example:

```
class Parent {
    protected int number = 10;  // Protected variable

    protected void display() {  // Protected method
        System.out.println("Number: " + number);
    }
}

class Child extends Parent {
    public void show() {
        System.out.println("Accessing from child class: " + number);  // Accessible in subclass
        display();  // Accessible in subclass
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Parent();
        // System.out.println(obj.number);  // Error: number has protected access
        // obj.display();  // Error: display() has protected access

        Child childObj = new Child();
        childObj.show();  // Allowed: Accessing from child class
    }
}
```

4. Default (Package-Private) Access Specifier

- **Visibility:** If no access specifier is mentioned, the member is accessible only within the same package.
- **Used for:** When you want the member to be accessible to other classes in the same package, but not from outside the package.

Example:

```
class MyClass {
    int number = 10;  // Default access (no specifier)

    void display() {  // Default access (no specifier)
        System.out.println("Number: " + number);
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        System.out.println(obj.number);  // Accessible within the same package
        obj.display();  // Accessible within the same package
    }
}
```

4. Explain call by value and call by reference with an example program

1. Call by Value

In Call by Value, when a method is called, t**he actual parameter (value passed) is copied into the formal parameter (local variable inside the method).** Changes made to the formal parameter inside the method do not affect the actual parameter.

Example of Call by Value:

```
class CallByValue {
    // Method to change value (does not affect the original value)
    public static void changeValue(int num) {
        num = num + 10;  // Modify the local variable
        System.out.println("Inside method, num: " + num); // 20
    }

    public static void main(String[] args) {
        int number = 10;  // Original value
        System.out.println("Before method call, number: " + number); // 10
        changeValue(number);  // Call by value
        System.out.println("After method call, number: " + number); // 10 (unchanged)
    }
}
```

Output

```
Before method call, number: 10
Inside method, num: 20
After method call, number: 10
```

2. Call by Reference

In Call by Reference, **the actual parameter is passed to the method as a reference (memory address).** Changes made to the parameter inside the method affect the original object outside the method.

Example of Call by Reference:

```
class CallByReference {
    // Method to change the value of an object (affects the original object)
    public static void changeValue(int[] arr) {
        arr[0] = 100;  // Modify the first element of the array
        System.out.println("Inside method, arr[0]: " + arr[0]); // 100
    }

    public static void main(String[] args) {
        int[] numbers = {10, 20, 30};  // Original array
        System.out.println("Before method call, arr[0]: " + numbers[0]); // 10
        changeValue(numbers);  // Call by reference (array is passed by reference)
        System.out.println("After method call, arr[0]: " + numbers[0]); // 100 (changed)
    }
}
```

Op:-
Before method call, arr[0]: 10
Inside method, arr[0]: 100
After method call, arr[0]: 100

5. Write a program to perform Stack operations using proper class and Methods.

```
import java.util.Scanner;

public class StackOperations {

    // Define Stack size
    static final int STACK_SIZE = 10;
    static int[] stack = new int[STACK_SIZE];
    static int top = -1; // Stack pointer

    // Push operation: Add an item to the stack
    public static void push(int item) {
        if (top == STACK_SIZE - 1) {
            System.out.println("Stack overflow");
        } else {
            stack[++top] = item;
            System.out.println(item + " pushed to stack");
        }
    }

    // Pop operation: Remove the top item from the stack
    public static void pop() {
        if (top == -1) {
            System.out.println("Stack underflow");
        } else {
            System.out.println("Item deleted = " + stack[top--]);
        }
    }

    // Display operation: Print all stack elements
    public static void display() {
        if (top == -1) {
            System.out.println("Stack is empty");
        } else {
            System.out.print("Stack: ");
            for (int i = 0; i <= top; i++) {
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }

    // Palindrome operation: Check if a string is a palindrome using stack
    public static void palindrome(String str) {
        top = -1;  // Reset top for new palindrome check

        // Push all characters of the string onto the stack
        for (int i = 0; i < str.length(); i++) {
            push(str.charAt(i));  // Push each character
        }

        // Compare characters of the string with the ones popped from the stack
        for (int i = 0; i < str.length(); i++) {
            if (str.charAt(i) != stack[top--]) {
                System.out.println(str + " : Is not a Palindrome");
                return;
```

```
            }
        }

        System.out.println(str + " : Is a Palindrome");
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int choice, item;
        String str;

        while (true) {
            System.out.println("1. Push");
            System.out.println("2. Pop");
            System.out.println("3. Display");
            System.out.println("4. Check if Palindrome");
            System.out.println("5. Exit");
            System.out.print("Enter your choice: ");
            choice = sc.nextInt();
            sc.nextLine();  // Consume the newline character

            switch (choice) {
                case 1:
                    System.out.print("Enter item to push: ");
                    item = sc.nextInt();
                    push(item);
                    break;

                case 2:
                    pop();
                    break;

                case 3:
                    display();
                    break;

                case 4:
                    System.out.print("Enter a string: ");
                    str = sc.nextLine();
                    palindrome(str);
                    break;

                case 5:
                    System.out.println("Exiting program...");
                    sc.close();
                    System.exit(0);
                    break;

                default:
                    System.out.println("Invalid choice, please try again.");
            }
        }
    }
}
```

6. Explain the use of this in JAVA with an example.

In Java, **this is a keyword that refers to the current object.** It is used to **refer to the current instance of a class,** especially when there is a need to differentiate between instance variables and method parameters that have the same name.

Uses of this Keyword:

1. **Referring to instance variables:**
   - When local variables or method parameters have the same name as instance variables, the this keyword is used to refer to instance variables.
2. **Invoking current class methods:**
   - this can be used to invoke current class methods.
3. **Passing the current object as a parameter:**
   - this can be passed as an argument to other methods or constructors.
4. **Invoking the current class constructor:**
   - The this() constructor can be used to call another constructor of the same class.

Example

```
class Car {
    // Instance variables
    String model;
    int year;

    // Constructor with parameters
    public Car(String model, int year) {
        // 'this' refers to the instance variables of the current object
        this.model = model;
        this.year = year;
    }

    // Method to display car details
    public void displayDetails() {
        System.out.println("Car Model: " + this.model);
        System.out.println("Car Year: " + this.year);
    }

    public static void main(String[] args) {
        // Create a new Car object
        Car myCar = new Car("Toyota", 2020);

        // Display the car details
        myCar.displayDetails();
    }
}
```

Op:-

```
Car Model: Toyota
Car Year: 2020
```

7. Explain java garbage collection mechanism by classifying 3 generations of java heap

Garbage collection is the process by which Java **automatically reclaims memory for objects that are no longer in use.**

• **Purpose:** Free up memory by removing unreferenced objects.
• **How:** Java uses the **gc()** method to suggest garbage collection.

Example code:

```
class Car {
protected void finalize() {
System.out.println("Object is garbage collected");
}
}
public class Main {
public static void main(String[] args) {
Car car = new Car();
car = null;
System.gc();
}
}
```

Output:

Object is garbage collected

3 Generations of the Java Heap

The Java Heap is divided into three parts to manage memory efficiently:

1. Young Generation
2. Old Generation (Tenured Generation)
3. Permanent Generation / Metaspace (Java 8 and beyond)

1. Young Generation:

- **Purpose:** Stores newly created objects.
  - **Components:**
- **Eden Space:** Where new objects are created.
- **Survivor Spaces (S0 and S1):** Objects that survive a GC cycle are moved here.
- **Garbage Collection:**
- **Minor GC:** Garbage collection happens here frequently. It cleans up the Young Generation and moves surviving objects to the Old Generation.
- **Note:** This process is fast and happens often.

2. Old Generation (Tenured Generation):

- **Purpose:** Stores long-lived objects that survive multiple GC cycles in the Young Generation.
- **Garbage Collection:**
- **Major GC (Full GC):** When the Old Generation fills up, a Major GC happens. It's slower because it collects garbage across the entire heap (both Young and Old Generations).

3. Permanent Generation / Metaspace (Java 8 and beyond):

- **Purpose:** Stores class metadata (information about classes, methods, etc.).
- **Java 7 and earlier:** This was the Permanent Generation.
- **Java 8 onwards:** It's now called Metaspace, which grows dynamically based on available system memory.
- **Garbage Collection:**
- **Metaspace GC:** Cleans up class metadata.

GC Process Simplified:

1. Young Generation: Objects are created. After Minor GC, surviving objects move to the Old Generation.
2. Old Generation: Objects that survive multiple cycles in the Young Generation are stored here. If full, a Major GC occurs.
3. Permanent/Metaspace: Holds class metadata. Metaspace is dynamically managed in Java 8+.

8. Develop a Java program to find area of rectangle are triangle using method over loading concept. Call these methods from main method with suitable inputs.

```
class Shape {

    // Method to calculate area of rectangle
    public double area(double length, double width) {
        return length * width;  // Formula for rectangle: length * width
    }

    // Method to calculate area of triangle
    public double area(double base, double height) {
        return 0.5 * base * height;  // Formula for triangle: 0.5 * base * height
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape = new Shape();

        // Rectangle: length = 5, width = 3
        double rectangleArea = shape.area(5, 3);
        System.out.println("Area of Rectangle: " + rectangleArea);

        // Triangle: base = 4, height = 6
        double triangleArea = shape.area(4, 6);
        System.out.println("Area of Triangle: " + triangleArea);
    }
}
```

Op:-

Area of Rectangle: 15.0
Area of Triangle: 12.0

9. Explain Methods in Java

i) Method with Parameters

A method **that accepts values as inputs.** Parameters allow data to be passed to the method.

• Helps in reusing the same method with different inputs.
• Improves modularity.

Example code:

```
class Calculator {
void sum(int a, int b) {
System.out.println("Sum: " + (a + b));
}
}
public class Main {
public static void main(String[] args) {
Calculator calc = new Calculator();
calc.sum(5, 10);
}
}
```

Output:
Sum: 15

ii) Method without Parameters

A method that **doesn't accept any input values.**
• Useful when the task is fixed and doesn't need any inputs.

Example:

```
class Greeter {
void greet() {
System.out.println("Hello!");
}
}
public class Main {
public static void main(String[] args) {
Greeter greetObj = new Greeter();
greetObj.greet();
}
}
```

Output: Hello!

iii) Method that Returns a Value

A method **that returns a value using the return keyword.**
• Helps in processing and sending back a result.
• The return type of the method must match the type of data being returned.

Example code:

```
class Calculator {
int multiply(int a, int b) {
return a * b;
}
}
public class Main {
public static void main(String[] args) {
Calculator calc = new Calculator();
int result = calc.multiply(5, 10);
System.out.println("Multiplication: " + result);
}
}
```

Output:

```
Multiplication: 50
```

10. Interpret the general form of a class with example.

A **class in Java is a blueprint for creating objects.** It defines the properties (variables) and behaviors (methods) that the objects created from the class will have.

Here's the general form of a class in Java:

```
class ClassName {

    // Instance variables (attributes)
    dataType variableName;

    // Constructor(s)
    ClassName() {
        // Initialization code
    }

    // Methods (behaviors)
    returnType methodName(parameters) {
        // Method body
    }
}
```

1. **class ClassName:**
   - The class keyword defines a class in Java, and ClassName is the name of the class.
   - Class names should start with an uppercase letter and follow CamelCase convention.
2. **Instance Variables (Attributes):**
   - Variables defined within the class that hold data related to the class.
   - Each object of the class will have its own copy of these variables.
3. **Constructor(s):**
   - A constructor is a special method used to initialize objects.
   - It has the same name as the class and does not have a return type.
   - A constructor can be parameterized or default (with no arguments).
4. **Methods (Behaviors):**
   - Methods define the behavior or actions that an object of the class can perform.
   - Each method has a return type (could be void if it doesn't return a value) and parameters (optional).

Example code:

```
class Car {
String model;
int year;
Car(String model, int year) {
```

```
this.model = model;
this.year = year;
void displayInfo() {
System.out.println("Model: " + model + ", Year: " + year);
}
}
}
public class Main {
public static void main(String[] args) {
Car car = new Car("Honda Civic", 2020);
car.displayInfo();
}
}
```

Output:

Model: Honda Civic, Year: 2020

11. Static Variable & Static Method

Static Variable

A static variable is shared among all instances of a class.
• Belongs to the class, not to individual objects.
• Memory allocation happens only once when the class is loaded.
• All objects of the class share the same static variable.

Example code:

```
class Car {
static int carCount = 0; // static variable
String model;
Car(String model) {
this.model = model;
carCount++;
void displayCar() {
System.out.println("Model: " + model);
}
}
}
public class Main {
public static void main(String[] args) {
Car car1 = new Car("Honda");
Car car2 = new Car("Toyota");
System.out.println("Total Cars: " + Car.carCount); // Accessing static variable
}
}
```

Output:

Total Cars: 2

Static Method

A static method belongs to the class rather than to instances of the class.
• Can be called without creating an object of the class.
• Can only access static variables directly.
• Used for utility or helper methods that don't need to access instance variables.

Example code:

```
class Calculator {
static int add(int a, int b) { // static method
return a + b;
}
}
public class Main {
public static void main(String[] args) {
int sum = Calculator.add(5, 10); // calling static method without object
System.out.println("Sum: " + sum);
}
}
```

Output: Sum: 15

12. Nested and Inner Class in Java

Nested Class

A nested class is a **class defined inside another class.**
• It logically groups classes that are only used in one place.
• Nested classes can be static or non-static.

Types of Nested Classes:

1. Static Nested Class:
• Declared with the static keyword.
• Can access static members of the outer class.

2. Inner Class (Non-static Nested Class):
• Not declared as static.
• Has access to all members (both static and non-static) of the outer class.

```
public class Main {
public static void main(String[] args) {
OuterClass.NestedClass nested = new OuterClass.NestedClass();
nested.displayMessage();
}
}
```

Example of Static Nested Class:

```
class OuterClass {
static String message = "Hello from Outer Class";
static class NestedClass {
void displayMessage() {
System.out.println(message); // Can access static variables of outer class
}
}
}
```

Output:
Hello from Outer Class

Example of Inner Class:

```
class OuterClass {
String message = "Hello from Outer Class";
class InnerClass {
void displayMessage() {
```

```
System.out.println(message); // Can access non-static variables of outer class
}
}
}
public class Main {
public static void main(String[] args) {
OuterClass outer = new OuterClass();
OuterClass.InnerClass inner = outer.new InnerClass();
inner.displayMessage();
}
```

Output:

```
Hello from Outer Class
```

13. static Keyword in Java(static variables and methods)

The **static keyword** is used to **declare class-level members** (variables and methods) that can be accessed without creating an instance of the class. It is shared by all instances of the class.

Usage of static:

- **Static variables:** Shared by all objects of the class.
- **Static methods:** Can be called without creating an object of the class.

Static Variable

A static variable is shared among all instances of a class.
• Belongs to the class, not to individual objects.
• Memory allocation happens only once when the class is loaded.
• All objects of the class share the same static variable.

Example code:

```
class Car {
static int carCount = 0; // static variable
String model;
Car(String model) {
this.model = model;
carCount++;
void displayCar() {
System.out.println("Model: " + model);
}
}
}
public class Main {
public static void main(String[] args) {
Car car1 = new Car("Honda");
Car car2 = new Car("Toyota");
System.out.println("Total Cars: " + Car.carCount); // Accessing static variable
}
}
```

Output:

```
Total Cars: 2
```

Static Method

A static method belongs to the class rather than to instances of the class.
• Can be called without creating an object of the class.
• Can only access static variables directly.
• Used for utility or helper methods that don't need to access instance variables.

Example code:

```
class Calculator {
static int add(int a, int b) { // static method
return a + b;
}
}
public class Main {
public static void main(String[] args) {
int sum = Calculator.add(5, 10); // calling static method without object
System.out.println("Sum: " + sum);
}
}
```

Output: Sum: 15

14. Method Overloading

Method overloading **allows methods with the same name but different parameters** to exist in the
same class.

i) Overloading by Number of Parameters

• Methods differ by the number of parameters.

Example code:

```
class Calculator {
void add(int a, int b) {
System.out.println("Sum: " + (a + b));
}
}
void add(int a, int b, int c) {
System.out.println("Sum: " + (a + b + c));
}
public class Main {
public static void main(String[] args) {
Calculator calc = new Calculator();
calc.add(5, 10);
calc.add(5, 10, 15);
}
}
```

Output:

```
Sum: 15
Sum: 30
```

ii) Overloading by Data Type

• Methods differ by the data type of parameters.

Example:

```
class Calculator {
void add(int a, int b) {
System.out.println("Sum (int): " + (a + b));
}
}
void add(double a, double b) {
System.out.println("Sum (double): " + (a + b));
}
public class Main {
public static void main(String[] args) {
Calculator calc = new Calculator();
calc.add(5, 10);
```

```
calc.add(5.5, 10.5);
}}
```

Op:

```
Sum (int): 15
Sum (double): 16.0
```

iii) Overloading by Sequence of Parameters

• Methods differ by the order of parameters.

Example:

```
class Calculator {
void display(int a, double b) {
System.out.println("Int and Double");
}
void display(double a, int b) {
System.out.println("Double and Int");
}
}
public class Main {
public static void main(String[] args) {
Calculator calc = new Calculator();
calc.display(5, 10.5);
calc.display(5.5, 10);
}
}
```

Output:

```
Int and Double
Double and Int
```

15. Constructor Overloading

Constructor overloading allows multiple constructors with different parameter lists in the same
class

Example code:

```
class Car {
String model;
int year;
// Default constructor
Car() {
this.model = "Unknown";
this.year = 2020;
}
// Parameterized constructor
Car(String model, int year) {
this.model = model;
this.year = year;
void displayInfo() {
System.out.println("Model: " + model + ", Year: " + year);
}
}
}
public class Main {
public static void main(String[] args) {
Car car1 = new Car();
Car car2 = new Car("Ford Mustang", 2021);
car1.displayInfo();
car2.displayInfo();
}
}
```

Output:

Model: Unknown, Year: 2020
Model: Ford Mustang, Year: 2021

16. Constructors in Java

i) Automatic Constructor
• If no constructor is provided, Java automatically creates a default constructor.
• This constructor has no parameters and does nothing special other than creating an object.

ii) Default Constructor
• A constructor with no parameters.
• Initializes the object with default or initial values.

Example code:

```
class Car {
String model;
Car() {
model = "Unknown Model";
} } }
void displayModel() {
System.out.println("Model: " + model);
public class Main {
public static void main(String[] args) {
Car car = new Car();
car.displayModel();
}
```

Output:
Model: Unknown Model

iii) Parameterized Constructor
• Constructor that accepts parameters to initialize object properties.
Example code:

```
class Car {
String model;
int year;
Car(String model, int year) {
this.model = model;
this.year = year;
void displayInfo() {
System.out.println("Model: " + model + ", Year: " + year);
}
}
}
public class Main {
public static void main(String[] args) {
Car car = new Car("Toyota Corolla", 2019);
car.displayInfo();
}}
```

Output: Model: Toyota Corolla, Year: 2019