JAVA UNIT 2

1. What is OOP? Explain OOP features

Object-Oriented Programming (OOP) is a programming approach where programs are designed using **objects** that represent real-world entities. Each object contains data (variables) and behavior (methods). OOP helps in organizing code in a structured, reusable, and secure way.

Features of OOP:

1. **Class**
   A class is a blueprint or template used to create objects. It defines what data and methods the objects will have.
2. **Object**
   An object is an instance of a class. It represents a real-life entity and contains state, behavior, and identity.
3. **Encapsulation**
   Encapsulation means wrapping data and methods together into a single unit called a class. It helps in protecting data using access modifiers.
4. **Inheritance**
   Inheritance allows one class to acquire properties and methods of another class. It helps in code reusability and reducing duplication.
5. **Polymorphism**
   Polymorphism means one name with many forms. It allows methods to behave differently based on object type using overloading and overriding.
6. **Abstraction**
   Abstraction hides internal implementation details and shows only essential features. It improves simplicity and readability of code.

2. Define class and object with purpose, syntax and example

Class

A class is a user-defined blueprint or template from which objects are created. It defines properties (variables) and behaviors (methods) common to all objects.

Purpose of Class:
The purpose of a class is to create multiple objects with the same structure. It helps in organizing data and logic together.

Syntax:

```
class ClassName {
  dataType variable;
  void method() { }
}
```

Object

An object is an instance of a class and represents a real-life entity. It occupies memory and has its own state.

Purpose of Object:
The purpose of an object is to access class members and perform operations. Each object has unique values for variables.

Syntax:

```
ClassName obj = new ClassName();
```

Example:
A Box class defines width, height, and depth. Objects like box1 and box2 store different values for these variables.

3. Differentiate between class and object

1. **Definition**
   A class is a blueprint used to create objects. An object is an actual instance of a class.
2. **Nature**
   A class is a logical entity and does not occupy memory. An object is a physical entity and occupies memory.
3. **Creation**
   A class is created using the class keyword. An object is created using the new keyword.
4. **Usage**
   A class defines structure and behavior. An object is used to access variables and methods.
5. **Multiple Instances**
   One class can create many objects. Each object has its own copy of instance variables.
6. **Example**
   class Dog is a class. Dog d1 = new Dog() is an object.

4. What is constructor? Purpose, rules, and types with example

A **constructor** is a special method used to initialize objects when they are created. It runs automatically when an object is created.

Purpose of Constructor

The main purpose of a constructor is to initialize instance variables. It ensures the object is ready for use immediately.

Rules for Constructor

1. Constructor name must be same as class name.
2. It has no return type, not even void.
3. It is called automatically during object creation.

Types of Constructors

1. **Default Constructor**
   It has no parameters and assigns default values.
2. **Parameterized Constructor**
   It accepts parameters to initialize objects with specific values.
3. **Copy Constructor (User-defined)**
   It copies values from one object to another.

Example Program (All Types):

```
class Student {
  int id;
  String name;
  Student() { id = 0; name = "NA"; }
  Student(int i, String n) { id = i; name = n; }
  Student(Student s) { id = s.id; name = s.name; }
}
```

5. Differentiate between constructor and method

1. **Purpose**
   A constructor initializes an object when it is created. A method performs a specific operation.
2. **Name**
   Constructor name must be same as class name. Method name can be any valid identifier.
3. **Return Type**
   A constructor has no return type. A method must have a return type or void.
4. **Invocation**
   Constructor is called automatically. Method is called explicitly using object name.
5. **Execution**
   Constructor executes only once per object creation. A method can be called multiple times.
6. **Usage**
   Constructor sets initial values. Method works on data after object creation.

6. Explain this keyword. Give different use cases with example

The **this keyword** in Java is a reference variable that refers to the **current object** of the class. It is mainly used to differentiate between **instance variables and local variables** when they have the same name.

Use cases of this keyword:

1. **Referring to instance variables**
   When local variables have the same name as instance variables, this helps avoid confusion. It clearly tells the compiler that the variable belongs to the current object.
2. **Calling current class constructor**
   this() can be used to call another constructor of the same class. This avoids code duplication inside constructors.
3. **Passing current object as parameter**
   The this keyword can pass the current object to a method or constructor. This is useful when one object needs to send itself to another method.
4. **Returning current object**
   this can also be returned from a method. It is commonly used in method chaining.

Example Program:

```
class Student {
  int id;
  Student(int id) {
    this.id = id;
  }
}
```

7. What are access specifiers? Purpose and types with example

Access specifiers in Java define the **visibility and accessibility** of class members like variables and methods. They control where the data can be accessed from, helping in data protection.

Purpose of access specifiers:

Access specifiers help in **encapsulation** by restricting unauthorized access. They also improve code security and maintainability.

Types of access specifiers:

1. **public**
   Members declared public can be accessed from anywhere. It has the widest visibility.
2. **private**
   Private members can be accessed only within the same class. It is mainly used to hide sensitive data.
3. **protected**
   Protected members are accessible within the same package and subclasses. It supports inheritance.
4. **default (no modifier)**
   Default access allows members to be accessed only within the same package. It is used when no keyword is specified.

Example Program:

```
class Demo {
  public int a;
  private int b;
  protected int c;
  int d; // default
}
```

8. Explain different use cases of static keyword

The **static keyword** is used when a member belongs to the **class rather than objects**. Static members are shared among all objects of the class.

Use cases of static keyword:

1. **Static variables**
   Used to store common data shared by all objects. Example: college name or company name.
2. **Static methods**
   Used for utility or helper functions. They can be called without creating an object.
3. **Static blocks**
   Used to initialize static variables. They execute once when the class is loaded.
4. **Main method**
   The main() method is static so the JVM can call it without creating an object.
5. **Object counter**
   Static variables are used to count how many objects are created. Since the value is shared, it gives correct results.

9. Explain static variables and static methods with example, purpose and syntax

Static Variables : A **static variable** belongs to the class, not to any specific object. Only **one copy** of the static variable exists in memory.

Purpose : It is used to store shared data common to all objects. Changes made affect all objects.

Syntax: static dataType variableName;

Static Methods : A static method also belongs to the class. It can be called directly using the class name.

Purpose: It is mainly used for utility methods that do not depend on object data.

Syntax : static returnType methodName() { }

Example Program:

```
class Student {
  static String college = "ABC College";
  static void display() {
    System.out.println(college);
  }
}
```

10. Differentiate between static variable and instance variable

1. **Ownership**
   A static variable belongs to the class. An instance variable belongs to an object.
2. **Memory allocation**
   Static variables are allocated memory only once. Instance variables get memory for every object.
3. **Sharing**
   Static variables are shared by all objects. Instance variables are separate for each object.
4. **Access**
   Static variables are accessed using class name. Instance variables are accessed using object name.
5. **Lifetime**
   Static variables exist until the class is unloaded. Instance variables exist as long as the object exists.
6. **Usage**
   Static variables store common values. Instance variables store object-specific data.

11. Explain final keyword along with its use cases

The **final keyword** in Java is used to **restrict modification**. It helps make programs more secure, stable, and predictable by preventing changes.

Use cases of final keyword:

1. **Final Variable**
   A variable declared as final cannot be changed once initialized. It is mainly used to create constant values like fixed rates or limits.
2. **Final Method**
   A method declared final cannot be overridden by subclasses. This is used when we want the method behavior to remain the same.
3. **Final Class**
   A final class cannot be inherited by any other class. It prevents modification of the class structure.

Example Program:

```
final class Demo {
  final int x = 10;
  final void show() {
    System.out.println(x);
  }}
```

12. Explain inheritance and its types with diagrams

Inheritance is an OOP feature where one class acquires the properties and methods of another class. It helps in **code reusability** and **method sharing**.

Purpose of inheritance

Inheritance reduces code duplication and improves program organization. It also supports extensibility of existing code.

Types of inheritance in Java:

1. **Single Inheritance**
   One subclass inherits from one superclass.

   Animal

   |

   Dog

2. **Multilevel Inheritance**
   A class is derived from another derived class.

   Animal

   |

   Mammal

   |

   Dog

3. **Hierarchical Inheritance**
   Multiple subclasses inherit from one superclass.

   Animal

   / \

   Dog Cat

4. **Hybrid Inheritance**
   Combination of more than one type of inheritance (achieved using interfaces).

Java does not support multiple inheritance using classes to avoid ambiguity problems.

13. Explain single inheritance with program

Single inheritance occurs when one subclass inherits from only one superclass. The subclass can use methods and variables of the parent class.

Explanation

This type of inheritance is simple and easy to understand. It promotes code reuse and avoids complexity.

Program Example:

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

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

public class Main {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.eat();
    d.bark();
  }
}
```

Here, Dog inherits the eat() method from Animal. This shows single inheritance clearly.

14. Explain multilevel inheritance with program

Multilevel inheritance occurs when a class is derived from another derived class. This creates a chain of inheritance.

Explanation : Each subclass inherits features from its parent class. It allows sharing of functionality across multiple levels.

Program Example:

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

class Mammal extends Animal {
  void walk() {
    System.out.println("Mammal walks");
  }
}

class Dog extends Mammal {
  void bark() {
    System.out.println("Dog barks");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog d = new Dog();
    d.eat();
    d.walk();
    d.bark();
  } }
```

Here, Dog inherits methods from both Animal and Mammal.

15. Explain hierarchical inheritance with program

Hierarchical inheritance occurs when multiple subclasses inherit from a single superclass. Each subclass has its own behavior.

Explanation : The base class properties are shared among all subclasses. It helps represent real-world relationships clearly.

Program Example:

```
class Animal {
  void eat() {
    System.out.println("Animal eats");
  }
}
class Dog extends Animal {
  void bark() {
    System.out.println("Dog barks");
  }
}
class Cat extends Animal {
  void meow() {
    System.out.println("Cat meows");
  }
}
public class Main {
  public static void main(String[] args) {
    Dog d = new Dog();
    Cat c = new Cat();
    d.eat();
    d.bark();
    c.eat();
    c.meow();
  } }
```

Here, both Dog and Cat inherit from Animal, showing hierarchical inheritance.

16. Define polymorphism and its types with example

Polymorphism means **one name, many forms**. In Java, it allows the same method name to perform different actions depending on how it is used.

Types of Polymorphism:

1. **Compile-time Polymorphism (Method Overloading)**
   This type of polymorphism is resolved at compile time. The compiler decides which method to call based on method signature.
2. **Run-time Polymorphism (Method Overriding)**
   This type of polymorphism is resolved at runtime. The method call depends on the object being referred to.

Example Program:

```
class Shape {
  void draw() {
    System.out.println("Drawing shape");
  }
}
class Circle extends Shape {
  void draw() {
    System.out.println("Drawing circle");
  }
}
public class Main {
  public static void main(String[] args) {
    Shape s = new Circle();
    s.draw();
  }
}
```

Here, the draw() method shows different behavior at runtime, demonstrating polymorphism.

17. Explain method overloading with example program

Method overloading occurs when multiple methods have the **same name but different parameter lists**. The difference may be in number, type, or both.

Explanation:

Method overloading supports compile-time polymorphism. It makes programs easier to read and use because related operations share the same name.

Important points:

1. Overloaded methods must differ in parameters. Return type alone cannot differentiate methods.
2. Method call is decided at compile time by the compiler.

Example Program:

```
class MathDemo {
  void add(int a, int b) {
    System.out.println(a + b);
  }
  void add(double a, double b) {
    System.out.println(a + b);
  }
}
public class Main {
  public static void main(String[] args) {
    MathDemo m = new MathDemo();
    m.add(10, 20);
    m.add(5.5, 4.5);
  }
}
```

The same method name add() performs different operations based on parameters.

18. Explain method overriding with example program

Method overriding occurs when a subclass provides its own implementation of a method already defined in its superclass. The method signature must be the same.

Explanation:

Method overriding supports runtime polymorphism. The method to be executed is decided during program execution based on object type.

Important points:

1. Overriding requires inheritance.
2. The overridden method is called using a parent class reference.

Example Program:

```
class Animal {
  void sound() {
    System.out.println("Animal makes sound");
  }
}
class Dog extends Animal {
  void sound() {
    System.out.println("Dog barks");
  }
}
public class Main {
  public static void main(String[] args) {
    Animal a = new Dog();
    a.sound();
  }
}
```

Here, sound() of Dog is called at runtime, showing method overriding.

19. Differentiate between method overloading and method overriding

1. **Definition**
   Method overloading means same method name with different parameters. Method overriding means redefining a superclass method in subclass.
2. **Polymorphism type**
   Overloading supports compile-time polymorphism. Overriding supports run-time polymorphism.
3. **Inheritance requirement**
   Overloading does not require inheritance. Overriding requires inheritance.
4. **Method signature**
   In overloading, parameters must differ. In overriding, method signature must be same.
5. **Decision time**
   Overloading is resolved at compile time. Overriding is resolved at runtime.
6. **Usage**
   Overloading improves method readability. Overriding provides specific behavior in subclasses.

20. Write a program in Java to create a Student class. Instance variables of Student class are rollno, name, marks1, marks2. Create a parameterized constructor for the class. Write method "printMarksheet" in Student class for printing marksheet of a student. Create 2 objects of Student class and print the marksheet of the student in main method.

```
class Student {
  int rollno;
  String name;
  double marks1;
  double marks2;

  // Parameterized Constructor
  Student(int r, String n, double m1, double m2) {
    rollno = r;
    name = n;
    marks1 = m1;
    marks2 = m2;
  }
```

```
    // Method to print marksheet
    void printMarksheet() {
        System.out.println("----- Marksheet -----");
        System.out.println("Roll No: " + rollno);
        System.out.println("Name: " + name);
        System.out.println("Marks 1: " + marks1);
        System.out.println("Marks 2: " + marks2);
        System.out.println("Total Marks: " + (marks1 + marks2));
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {

        // Creating two Student objects
        Student s1 = new Student(1, "Riya", 85, 90);
        Student s2 = new Student(2, "Amit", 78, 88);

        // Printing marksheets
        s1.printMarksheet();
        s2.printMarksheet();
    }
}
```

21. Create a class BankAccount with data members: accountNumber, accountHolderName, and balance.Include a constructor to initialize values.Add methods: deposit(), withdraw(), and displayBalance().Write a main program to create objects for 2 customers and perform some transactions.

```
class BankAccount {
  int accountNumber;
  String accountHolderName;
  double balance;

  // Parameterized Constructor
  BankAccount(int accNo, String name, double bal) {
    accountNumber = accNo;
    accountHolderName = name;
    balance = bal;
  }

  // Method to deposit amount
  void deposit(double amount) {
    balance = balance + amount;
    System.out.println("Amount Deposited: " + amount);
  }

  // Method to withdraw amount
  void withdraw(double amount) {
    if (amount <= balance) {
      balance = balance - amount;
      System.out.println("Amount Withdrawn: " + amount);
    } else {
      System.out.println("Insufficient Balance");
    }
  }
```

```
    // Method to display balance
    void displayBalance() {
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Account Holder: " + accountHolderName);
        System.out.println("Current Balance: " + balance);
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {

        // Creating two BankAccount objects
        BankAccount b1 = new BankAccount(101, "Riya", 5000);
        BankAccount b2 = new BankAccount(102, "Amit", 8000);

        // Transactions for first customer
        b1.deposit(2000);
        b1.withdraw(1500);
        b1.displayBalance();

        // Transactions for second customer
        b2.withdraw(3000);
        b2.deposit(1000);
        b2.displayBalance();
    }
}
```

22. Define a class Book with data members: title, author, price.Write a parameterized constructor to initialize details.Write methods: displayDetails() and applyDiscount(double percent).Create an array of 3 Book objects and display their details.

```
class Book {
   String title;
   String author;
   double price;

   // Parameterized Constructor
   Book(String t, String a, double p) {
      title = t;
      author = a;
      price = p;
   }

   // Method to apply discount
   void applyDiscount(double percent) {
      price = price - (price * percent / 100);
   }

   // Method to display book details
   void displayDetails() {
      System.out.println("Title: " + title);
      System.out.println("Author: " + author);
      System.out.println("Price: " + price);
      System.out.println();
   }
}
```

```
public class Main {
  public static void main(String[] args) {

    // Creating array of Book objects
    Book[] books = new Book[3];

    books[0] = new Book("Java Basics", "James", 500);
    books[1] = new Book("OOP Concepts", "Ritu Jain", 600);
    books[2] = new Book("Data Structures", "Mark", 700);

    // Applying discount and displaying details
    for (int i = 0; i < books.length; i++) {
      books[i].applyDiscount(10); // 10% discount
      books[i].displayDetails();
    }
  }
}
```

23. Create a class Employee with data members: empId, name, basicSalary.Use a constructor to initialize details.Add a method calculateNetSalary() that computes net salary with DA = 20% and HRA = 10%.Create objects of 2 employees and display their salaries.

```
class Employee {
  int empId;
  String name;
  double basicSalary;
  // Parameterized Constructor
  Employee(int id, String n, double salary) {
    empId = id;
    name = n;
    basicSalary = salary;
  }
```

```
    // Method to calculate and display net salary
    void calculateNetSalary() {
        double da = basicSalary * 0.20;   // 20% DA
        double hra = basicSalary * 0.10;  // 10% HRA
        double netSalary = basicSalary + da + hra;

        System.out.println("Employee ID: " + empId);
        System.out.println("Employee Name: " + name);
        System.out.println("Basic Salary: " + basicSalary);
        System.out.println("Net Salary: " + netSalary);
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {

        // Creating two Employee objects
        Employee e1 = new Employee(101, "Riya", 30000);
        Employee e2 = new Employee(102, "Amit", 40000);

        // Displaying salaries
        e1.calculateNetSalary();
        e2.calculateNetSalary();
    }
}
```

24. Explain runtime polymorphism with example program

Runtime polymorphism (also called **dynamic polymorphism**) is a feature of OOP where the **method call is resolved at runtime**, not at compile time. In Java, runtime polymorphism is achieved using **method overriding**. This means a subclass provides its own implementation of a method that is already defined in its superclass, and the method to be executed depends on the **object type**, not the reference type.

Explanation (Point-wise):

1. **Method overriding is the base of runtime polymorphism**
   In runtime polymorphism, a subclass overrides a method of its parent class. The method signature (name and parameters) must be the same in both classes.
2. **Decision is made at runtime**
   The JVM decides which method to execute during program execution. This decision depends on the object being referred to, not the reference variable.
3. **Superclass reference can refer to subclass object**
   A parent class reference variable can store the reference of a child class object. This is required to achieve runtime polymorphism.
4. **Supports flexibility and extensibility**
   Runtime polymorphism allows different subclasses to provide different implementations of the same method. This makes programs more flexible and easier to extend.

Example Program:

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

class Dog extends Animal {
  void sound() {
    System.out.println("Dog barks");
  }
}
```

```
class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {

        Animal a1 = new Dog();  // parent reference, child object
        Animal a2 = new Cat();

        a1.sound();  // calls Dog's sound()
        a2.sound();  // calls Cat's sound()
    }
}
```