Java Architecture

1. Java Virtual Machine (JVM)

The **JVM (Java Virtual Machine)** is the core component of the Java Runtime Environment (JRE). It provides a runtime environment that executes Java bytecode and ensures platform independence through the **Write Once, Run Anywhere (WORA)** principle.

JVM Components:

1. **Class Loader** - Loads compiled .class files into memory, handles dynamic linking, and manages namespaces.
2. **Runtime Memory Areas**:
   - **Method Area**: Stores class metadata, including static variables and method bytecode.
   - **Heap**: Allocates memory for objects and manages garbage collection.
   - **Stack**: Stores method-specific values such as local variables and function calls.
   - **PC Register**: Keeps track of the currently executing instruction.
   - **Native Method Stack**: Manages native method calls.
3. **Execution Engine** - Converts bytecode into machine code using:
   - **Interpreter** (Executes bytecode line by line, slower performance)
   - **JIT Compiler (Just-In-Time Compiler)** (Compiles frequently used bytecode into native machine code for better performance)
4. **Garbage Collector** - Automatically deallocates memory from unused objects, preventing memory leaks.

2. Java Runtime Environment (JRE)

The **JRE (Java Runtime Environment)** provides libraries, JVM, and other components required to run Java applications. It does not include development tools like compilers or debuggers. JRE is sufficient if you only need to run Java programs and not develop them.

Components of JRE:

- **JVM (Java Virtual Machine)**: Executes Java applications.
- **Java Class Libraries**: Predefined standard libraries that provide essential functionalities.
- **Java Launcher**: Starts the execution of Java programs.

3. Java Development Kit (JDK)

The **JDK (Java Development Kit)** is a complete development package that includes everything required to develop, compile, and run Java applications. It contains the JRE, development tools, and additional utilities.

Components of JDK:

- **JRE (Java Runtime Environment)**: Includes JVM and standard libraries.

- **Compiler (javac)**: Converts Java source code (.java) into bytecode (.class).
- **Debugger (jdb)**: Helps in debugging Java programs.
- **JavaDoc (javadoc)**: Generates documentation from Java source code comments.
- **Other Tools**: Includes monitoring, security, and packaging tools.

Java Code Execution Flow:

1. **Write Code**: HelloWorld.java
2. **Compile**: javac HelloWorld.java → Generates HelloWorld.class
3. **Run**: java HelloWorld

Object-Oriented Programming (OOP) Concepts

1. Class and Object

What is a Class?

A *class* in Java is a blueprint for creating objects. It defines the structure and behavior of objects by specifying variables and methods.

What is an Object?

An *object* is an instance of a class that holds the actual values and can invoke methods defined in the class.

Constructor

A constructor is a special method that is automatically called when an object is instantiated. It initializes object properties.

Types of Constructors:

1. **Default Constructor:** Automatically provided if no constructor is defined.
2. **Parameterized Constructor:** Used to assign specific values during object creation.

Example:

```
class Car { // Car class

  String model; // variable

  int year;   // variable
```

```
    Car(String m, int y) { // Constructor

        model = m; // Set model variable

        year = y;   // Set year variable

    }

    void display() { // Display info

        System.out.println(model + " (" + year + ")");

    }

    public static void main(String[] args) {

        Car myCar = new Car("Camry", 2023); // Create car object

        myCar.display(); // Show info

        Car yourCar = new Car("Civic", 2020); // Another car object

        yourCar.display(); // Show info

    }

}
```

2. Inheritance

What is Inheritance?

Inheritance is the ability of a class to get all the methods and variables from another class by using the extends keyword in the syntax to achieve inheritance. It enables code reuse, hierarchical classification, and polymorphism.

Types of Inheritance in Java:

1. **Single Inheritance** - A class inherits from one superclass.
2. **Multilevel Inheritance** - A class inherits from another class, forming a chain.
3. **Hierarchical Inheritance** - Multiple classes inherit from a single superclass.
4. **Hybrid Inheritance** - A combination of multiple inheritance types (achieved using interfaces).

Note: Java does not support ***Multiple Inheritance*** (one class extending multiple classes) to avoid ambiguity errors. The extends keyword does not allow multiple names after it.

Example:

```
class Employee {
    private String name;
    protected double salary;
    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
    public void displayDetails() {
        System.out.println("Name: " + name + ", Salary: $" + salary);
    }
}
```

Usage:

```
class Manager extends Employee {
    private String department;
    public Manager(String name, double salary, String department) {
        super(name, salary);
        this.department = department;
    }
    @Override
    public void displayDetails() {
        super.displayDetails();
        System.out.println("Department: " + department);
    }
}
```

3. Polymorphism

Polymorphism means "*many forms*", and it occurs when we have many classes that are related to each other by *inheritance*. It is the ability of a method, function, or operator to behave differently based on the context. It is classified into **Compile-time (Method Overloading) and Runtime (Method Overriding) Polymorphism.**

Compile-time Polymorphism (Method Overloading)

Defining multiple methods in the same class with the same name but different parameters (different number of parameters, different types of parameters, or both). The compiler determines which method to call based on the arguments provided at compile time. This is also known as *static binding* or *early binding*.

Example:

```
class MathOperations {

    int add(int a, int b) {

        return a + b;

    }

    double add(double a, double b) {

        return a + b;

    }

    int add(int a, int b, int c) {

        return a + b + c;

    }

}
```

Runtime Polymorphism (Method Overriding)

When a subclass (child class) provides a specific implementation for a method that is already defined in its superclass (parent class), it's called method overriding. The method in the subclass "overrides" the method in the superclass. At runtime, the Java Virtual Machine (JVM) determines the actual type of the object and calls the appropriate version of the method. This is also known as *dynamic binding* or *late binding*.

Example:

```
class Parent {

    void display() {
```

```
    System.out.println("Display from Parent class");
  }
}

class Child extends Parent {
  @Override
  void display() {
    System.out.println("Display from Child class");
  }
}
```

4. Encapsulation

The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as *private* access specifier and provide *public* **get** and **set** methods to access and update the value of a private variable.

The **get** method returns the variable value, and the **set** method sets the value. Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case. This prevents direct, unauthorized access to data, ensuring data integrity. This is a major security benefit.

Example:

```
class BankAccount {
  private String accountHolder;
  private double balance;
  public BankAccount(String accountHolder, double balance) {
    this.accountHolder = accountHolder;
    this.balance = balance;
  }
  public String getAccountHolder() {
        return accountHolder;
    }
    public double getBalance() {
        return balance;
    }
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("John Doe", 1000.0);
        System.out.println("Account Holder: " + account.getAccountHolder());
        System.out.println("Balance: " + account.getBalance());
        account.deposit(500.0);
        System.out.println("Updated Balance: " + account.getBalance());
    }
}
```

5. Abstraction

The abstract class defines a general concept or a contract. It specifies *what* something should do, but not *how* it should do it. This is the "idea" part. Different classes can then get this abstract idea by *inheritance* and implement in their own way.

Abstraction is achieved using **abstract classes** or **interfaces** to define a blueprint for classes without providing full implementation. An **abstract class** is a class that cannot be instantiated and may contain both abstract methods (without implementation) and concrete methods (with implementation).

“abstract” (non-specifier) is the keyword used for abstraction, if a method/function in a class has no implementations and ends with termination (“;”) then the method sound be defined as an abstract and if a class contains one or more *abstract methods* then the class should be mentioned as *abstract class*. An abstract method must always be redefined in the subclass, thus making *overriding* compulsory or making the subclass itself abstract.

Example:

```
abstract class Animal {

  abstract void makeSound();

}



class Dog extends Animal {

  void makeSound() { System.out.println("Woof"); }

}
```

Usage:

```
public class MainAbstract {

  public static void main(String[] args) {

    Animal myDog = new Dog();

    myDog.makeSound(); // Output: Woof

  }

}
```

6. Interface

Another way to achieve *abstraction* in Java, is with *Interface*. It is a blueprint or a contract that specifies a set of methods that a class must implement. A class uses the “*implements*” keyword to indicate that it will provide implementations for the methods defined in an interface.

A class can implement multiple interfaces. This allows a class to inherit multiple behaviors or contracts, effectively achieving a form of *multiple inheritance*, which Java doesn't directly support with classes.

A class can extend another class, and similarly, an interface can extend another interface. However, only a class can implement an interface, and the reverse (an interface implementing a class) is not allowed.

Example:

```
interface Speaker {

   void speak();

}



class Human implements Speaker {

   public void speak() {

      System.out.println("Hello!");

   }

}



class Parrot implements Speaker {

   public void speak() {

      System.out.println("Squawk!");

   }

}
```

Usage:

```
public class MainInterface2 {

    public static void main(String[] args) {

        Speaker person = new Human();

        Speaker bird = new Parrot();



        person.speak(); // Output: Hello!

        bird.speak();   // Output: Squawk!

    }

}
```

Default Methods in Interfaces (Introduced in Java 8)

Before Java 8, interfaces could only have **abstract methods**. With Java 8, **default methods** were introduced, allowing methods with implementations inside interfaces. To create a method with implementations inside an interface "default" keyword should be mentioned in the method signature.

Use of Default Methods:

- To add new methods to existing interfaces **without breaking existing implementations**.
- Allows interfaces to have **some behavior** without forcing subclasses to override them.

Example:

```
interface Vehicle {

    void start(); // Abstract method (must be implemented by classes)



    default void fuelType() { // Default method with implementation

        System.out.println("Default Fuel: Petrol");

    }

}
```

```
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car started!");
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();     // Calls overridden method
        myCar.fuelType();  // Calls default method from interface
    }
}
```

Design Patterns

Singleton Design Pattern

The Singleton Design Pattern ensures a class has *only one* instance and provides a global access point to it. It's ideal for scenarios requiring centralized control, like managing database connections or configuration settings.

Example:

```
class Singleton {
    private static Singleton instance;
    private Singleton() {
        // Private constructor prevents instantiation
    }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    public void showMessage() {
        System.out.println("Singleton Instance Accessed!");
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();
        singleton.showMessage();
    }
}
```

Static

The static keyword in Java is used for **memory management** and allows methods or variables to belong to a class rather than an instance. It helps in reducing memory usage and improving performance.

1. Static Variables

- A static variable is shared across all instances of a class.
- It is initialized only once at the class level.

Example:

```
class Employee {
    static String company = "Tech Corp"; // Shared variable

}

public class Main {

    public static void main(String[] args) {

        System.out.println(Employee.company); // No need to create an object

    }

}
```

2. Static Methods

- Static methods belong to the class rather than an object.
- They cannot access non-static (instance) variables directly.

Example:

```
class MathUtils {

    static int square(int num) {

        return num * num;

    }

}

public class Main {

    public static void main(String[] args) {

        System.out.println(MathUtils.square(5)); // Directly calling static method

    }

}
```

3. Static Blocks

- A static block is executed **once** when the class is loaded.
- It is useful for **initializing static variables**.

Example:

```
class Demo {
    static {
        System.out.println("Static block executed!");
    }
}
public class Main {
    public static void main(String[] args) {
        new Demo(); // Class is loaded, static block runs
    }
}
```

Final

The final keyword in Java is used to **restrict modification**. It can be applied to **variables, methods, and classes**.

1. Final Variable (Constant Value)

A **final variable** cannot be changed once assigned. It must be initialized at declaration or inside a constructor.

Example:

```
class Constants {
    final int MAX_SPEED = 120; // Constant variable
    void display() {
        // MAX_SPEED = 150;  // Compilation error: Cannot modify final variable
        System.out.println("Max Speed: " + MAX_SPEED);
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Constants obj = new Constants();
        obj.display();
    }
}
```

2. Final Method (Prevent Overriding)

A **final method cannot be overridden** by subclasses. However, it can still be inherited.

Example:

```
class Parent {
    final void show() {
        System.out.println("Final method in Parent class");
    }
}


class Child extends Parent {
    // void show() { } // Compilation error: Cannot override final method
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.show(); // Calls the inherited final method
    }
}
```

3. Final Class (Prevent Inheritance)

A **final class cannot be extended** (inherited) by any other class.

Example:

```
final class Vehicle {
    void show() {
        System.out.println("This is a final class");
    }
}


// class Car extends Vehicle { } // Compilation error: Cannot inherit from final class
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        Vehicle obj = new Vehicle();
        obj.show();
    }
}
```

Access Specifiers in Java

Access specifiers (also known as **Access Modifiers**) control the visibility of **classes, methods, and variables** in Java.

Types of Access Specifiers

Modifier	**Scope**
public	Accessible from anywhere in the program.
private	Accessible only within the same class.
protected	Accessible within the same package and by subclasses.
(default)	Accessible only within the same package (if no modifier is specified).