UNIT - 2

misc

Introduction to Object Oriented Programming

Java was mainly developed to create software for consumer electronic devices that could be controlled by a remote

Why Use Java?
• Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
• It is one of the most popular programming languages
• It is easy to learn and simple to use
• It is open-source and free
• It is secure, fast and powerful
• It has huge community support (tens of millions of developers)
• Java is an object-oriented language that gives a clear structure to programs and allows code to be
reused, lowering development costs

Features of Java and their Justifications:

i) **High Performance:**

- **Justification:** Java achieves high performance through Just-In-Time (JIT) compilation. The bytecode is initially interpreted by the JVM, and frequently

executed code paths are identified. These code paths are then compiled into native machine code for direct execution by the hardware. This JIT compilation process improves the execution speed of Java programs, approaching the performance of natively compiled languages.

ii) **Portable:**

- **Justification:** Java's portability is mainly due to the Java Virtual Machine (JVM). Java programs are compiled into platform-independent bytecode, which can be executed on any device or operating system with a compatible JVM. Since the JVM abstracts away the hardware and operating system dependencies, Java programs can run unchanged on various platforms, making Java highly portable.

iii) **Robust:**

- **Justification:** Java provides several features that contribute to its robustness:
  - **Strong Memory Management:** Java's memory management system, including automatic garbage collection, prevents memory leaks and helps ensure that Java programs use memory efficiently without crashing due to memory-related issues.
  - **Exception Handling:** Java's robust exception handling mechanism allows developers to catch and handle runtime exceptions gracefully, preventing programs from crashing unexpectedly.
  - **Type Checking:** Java is a statically typed language, which means that type checking is performed at compile time. This helps catch many errors early in the development process, reducing the likelihood of runtime errors and making Java programs more robust.
  - **No Pointers:** Java does not support pointer arithmetic, which helps prevent many common programming errors such as buffer overflows, memory corruption, and security vulnerabilities.

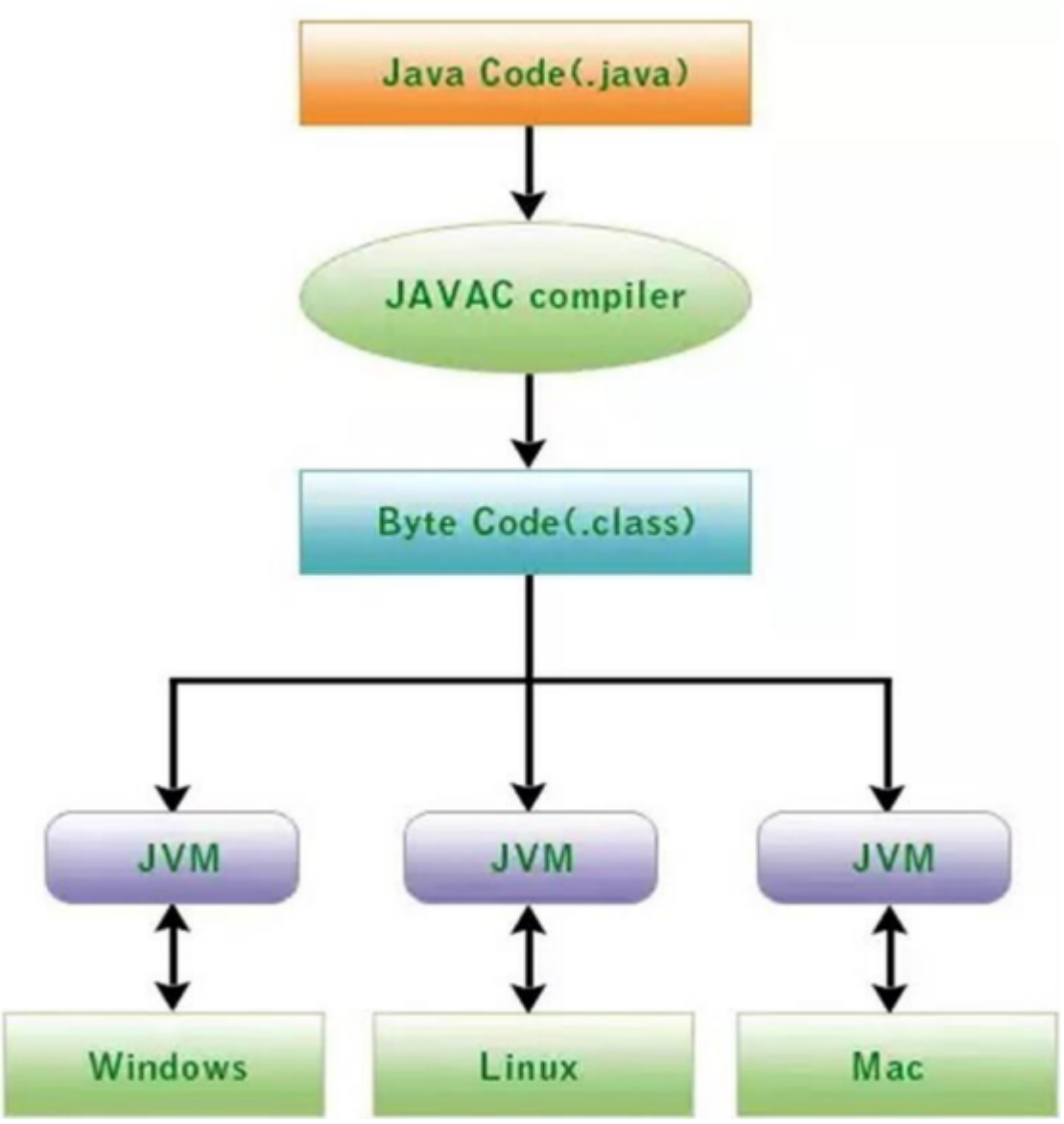
Java Uses

Mobile applications (specially Android apps), Desktop applications, Web applications
Web servers and application servers, Games, Database connection.

Some common terms in Java programming include:

1. **Java Virtual Machine (JVM)**: Executes Java bytecode. It provides a runtime environment for Java programs, enabling them to be platform-independent. JVMs are available for various hardware and software platforms.
2. **Java Development Kit (JDK)**: A comprehensive development kit for Java, which includes the compiler (javac), Java Runtime Environment (JRE), debuggers, documentation (JavaDocs), and other tools necessary for Java development.
3. **Java Runtime Environment (JRE)**: A subset of the JDK, JRE includes the JVM along with libraries and other components required for running Java applications. It does not include development tools like compilers.
4. **Bytecode**: Compiled Java code is transformed into bytecode, which is platform-independent intermediate code. This bytecode is executed by the JVM.
5. **Compiler**: Converts Java source code into bytecode that can be executed by the JVM. The primary Java compiler is 'javac', included in the JDK.
6. **Applet**: A small Java program that runs within a web browser. They were popular in the early days of the internet for creating interactive web content but have largely been replaced by other technologies.
7. **Plugin**: A software component that adds specific features or functionality to a larger software application. In the context of Java, plugins can extend the capabilities of Java applications or web browsers.
8. **Browser**: Software used to access and view information on the World Wide Web. JRE includes components necessary for Java applets to run within a browser.
9. **Execution phases**: The stages a Java program goes through, including writing code, compiling it into bytecode, and running the bytecode on the JVM.
10. **Debuggers**: Tools used for finding and fixing errors (bugs) in software code. Java provides debugging tools as part of the JDK for diagnosing and troubleshooting issues in Java programs.

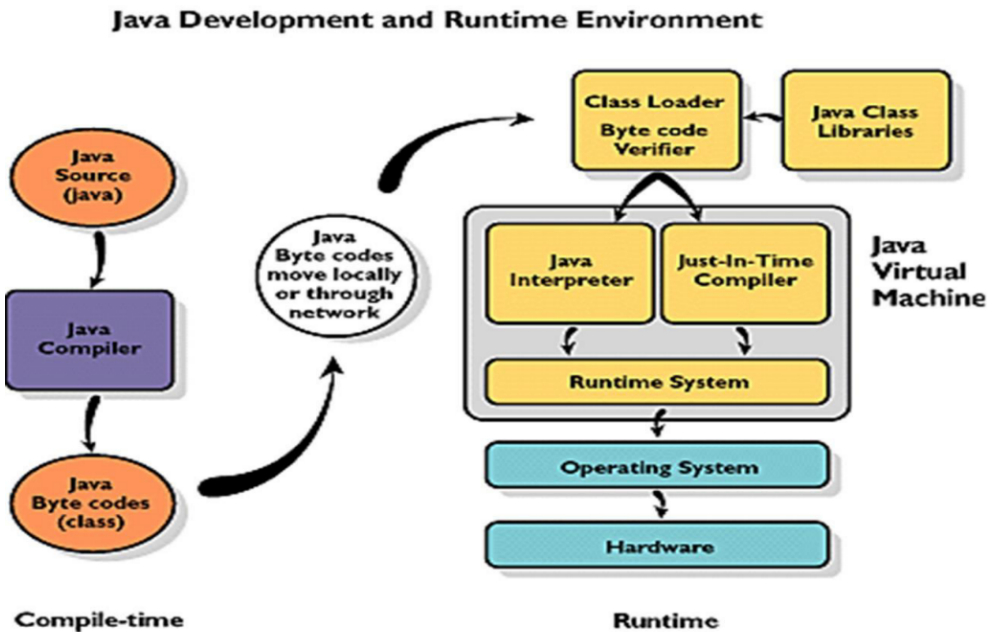
JVM



1. **Specification**: The JVM is defined by a specification that outlines how it should function. This specification ensures that Java bytecode can be executed consistently across different platforms. While the specification defines the behavior of the JVM, the implementation details, such as the algorithms used, can vary between different providers. Oracle and other companies provide implementations of the JVM.
2. **Implementation**: The actual software that interprets Java bytecode and executes it is called the Java Runtime Environment (JRE). The JRE includes the JVM along with libraries and other components necessary for running Java applications. It is responsible for loading and executing Java bytecode.

3. **Runtime Instance**: Whenever you run a Java program using the `java` command on the command prompt, an instance of the JVM is created. This instance loads the bytecode of the Java class and executes it. Each time you run a Java program, a new instance of the JVM is created to handle the execution.

Java Runtime Environment(JRE)

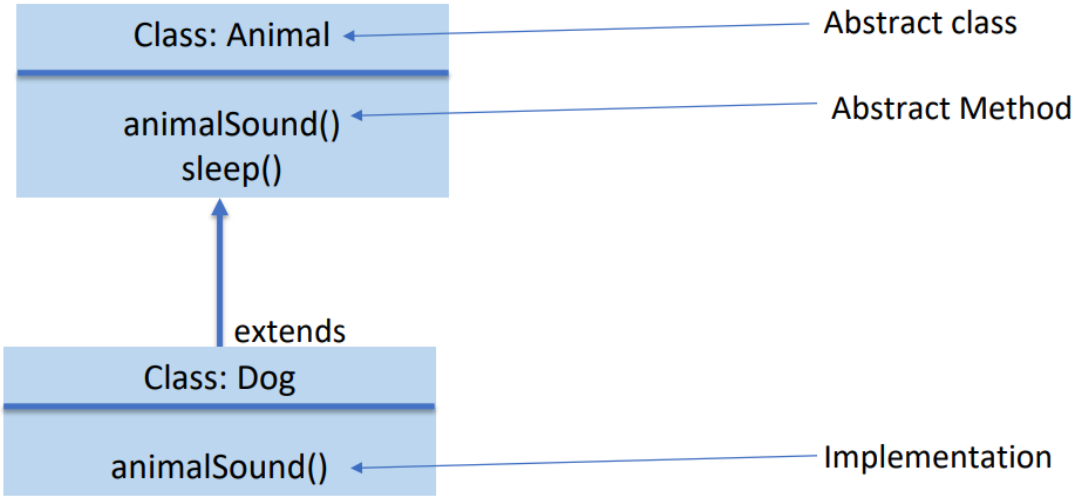


Abstraction

1. **Definition of Abstraction**: Abstraction involves providing users with functionality while hiding implementation details. Users interact with entities based on what they do rather than how they are implemented internally.
2. **Abstraction in Java**: Java achieves abstraction through interfaces and abstract classes. Interfaces allow for complete abstraction, while abstract classes serve as blueprints for other classes and can contain both abstract and concrete methods.
3. **Advantages of Java Abstraction**:
    - Reduces complexity by focusing on essential features.

- Avoids code duplication by promoting reusable code through interfaces and abstract classes.
- Eases maintenance as changes to implementation details don't affect the user interface.
- Increases security and confidentiality by hiding sensitive information.

4. **Abstract Class in Java**:
   - An abstract class is a class that cannot be instantiated, meaning objects cannot be created directly from it.
   - Declared using the `abstract` keyword and represents a concept rather than a concrete implementation.
   - Can contain both abstract methods (without body) and non-abstract methods (with body).
   - If a class contains at least one abstract method, it must be declared abstract itself.
   - Objects of an abstract class cannot be instantiated, but its subclasses can be used.
   - Allows for achieving varying levels of abstraction, from 0 to 100%.
   - Can include constructors, including parameterized constructors, and may also contain final and static methods.



```
abstract class Animal { // Abstract class
    public abstract void animalSound(); // Abstract meth
od (does not have a body)

    public void sleep() { // Regular method
        System.out.println("Zzz");
    }
}

class Dog extends Animal { // Subclass (inherits from An
imal)
    public void animalSound() {
        System.out.println("The Dog says: bow bow "); //
The body of animalSound() is provided here
    }
}

class Main {
    public static void main(String[] args) {
        Dog mydog = new Dog(); // Create a Dog object
        mydog.animalSound();
        mydog.sleep();
    }
}
```

In this example:

- `Animal` is an abstract class that contains one abstract method `animalSound()` and one regular method `sleep()`.
- `Dog` is a subclass of `Animal`. It provides the implementation for the abstract method `animalSound()`.
- In the `Main` class, a `Dog` object `mydog` is created and its `animalSound()` and `sleep()` methods are called.

Output:

```
The Dog says: bow bow
```

```
Zzz
```

This example demonstrates how abstraction allows us to define a common interface (`Animal`) with abstract methods that concrete subclasses (`Dog`) implement. The `Animal` class defines what each animal should be able to do (`animalSound()`), while leaving the specific implementation details to the subclasses. This promotes code reusability and flexibility in the design.

Encapsulation

Encapsulation is a fundamental concept in object-oriented programming that involves bundling data and methods that operate on that data into a single unit. Here's a breakdown of the key points regarding encapsulation:

Definition:

- Encapsulation is the process of wrapping up data (variables) and methods (functions) into a single unit, typically referred to as a class in object-oriented programming.
- The data within the encapsulated unit is not directly accessible from outside. Instead, access is provided through methods that act as interfaces to the object's data.

Advantages of Encapsulation:

1. **Data Hiding:** By encapsulating data, implementation details are hidden from the outside world, which helps in protecting the integrity of the data.
2. **Increased Flexibility:** Encapsulation allows for controlling access to data, making variables read-only or write-only as needed, thereby increasing flexibility.
3. **Reusability:** Encapsulation promotes reusability by providing a clear and well-defined interface for interacting with objects, making it easier to reuse code in different parts of a program or in different programs altogether.
4. **Testing Code:** Encapsulated code is easier to test, particularly for unit testing, as the behavior of the class can be tested independently of its internal implementation details.

5. **Freedom in Implementation:** Encapsulation provides freedom to programmers in implementing the details of a system, as the internal workings of the class can be changed without affecting other parts of the program.

Disadvantages of Encapsulation:

1. **Increased Complexity:** Improper use of encapsulation can lead to increased complexity, especially if the encapsulation boundaries are not well-defined or if the encapsulated methods become overly complex.
2. **Reduced Understandability:** In some cases, encapsulation can make it more difficult to understand how the system works, especially if the encapsulated class has complex interactions with other parts of the system.
3. **Limitation on Flexibility:** While encapsulation increases flexibility in some aspects, it may also limit flexibility in others, particularly if the encapsulated methods restrict access to certain data or functionality.

In the provided example, encapsulation is demonstrated through the `Person` class, where the data member `name` is encapsulated within the class using private access modifier. This prevents direct access to the `name` variable from outside the class, ensuring data hiding.

Here's the breakdown:

```
class Person {
    private String name; // private = restricted access

    public String getName() { // Getter
        return name;
    }

    public void setName(String newName) { // Setter
        this.name = newName;
    }
}

class Main {
    public static void main(String[] args) {
        Person myObj = new Person();
```

```
        myObj.name = "John"; // error - Attempting to ac
cess private variable
        System.out.println(myObj.name); // error - Attem
pting to access private variable
    }
}
```

Explanation:

- In the `Person` class, the `name` variable is declared with the `private` access modifier, making it inaccessible from outside the class.
- Getter and setter methods (`getName()` and `setName()`) are provided to access and modify the `name` variable, respectively. These methods provide controlled access to the private variable.
- In the `Main` class, when attempting to directly access the `name` variable using `myObj.name`, compilation errors occur because `name` is private and cannot be accessed from outside the `Person` class.
- Instead, to access or modify the `name` variable, one must use the provided getter and setter methods (`getName()` and `setName()`).

Output:

```
ERROR!
```

The errors indicate that direct access to the `name` variable outside the `Person` class is not permitted due to its private access modifier, thus demonstrating encapsulation by enforcing data hiding.

```
import java.util.ArrayList;

class Teacher {
    private String name;
    private String subject;

    public Teacher(String name) {
        this.name = name;
    }
```

```
    public Teacher(String name, String subject) {
        this.name = name;
        this.subject = subject;
    }

        public String getName() {
        return name;
    }

    public String getSubject() {
        return subject;
    }
}

class Anchor extends Teacher {
    private ArrayList<Teacher> coTeachers;

    public Anchor(String name) {
        super(name);
        coTeachers = new ArrayList<>();
    }

    public Anchor(String name, String subject) {
        super(name, subject);
        coTeachers = new ArrayList<>();
    }

    public void addTeacher(Teacher teacher) {
        coTeachers.add(teacher);
    }

    public void showDetails() {
        System.out.println("Anchor Name: " + getName());
        System.out.println("Anchor Subject: " + getSubject
        System.out.println("Co-teachers:");
        for (Teacher teacher : coTeachers) {
            System.out.println("- " + teacher.getName() +
        }
```

```
    }
}

public class MainClass {
    public static void main(String[] args) {
        Teacher t1 = new Teacher("Teacher Name");
        Teacher t2 = new Teacher("Teacher Name", "Teacher 
        Anchor a1 = new Anchor("Teacher Name", "Teacher Su
        a1.addTeacher(t1);
        a1.addTeacher(t2);
        a1.showDetails(); // shows details of Anchor as we
    }
}
```

Composition

Composition in object-oriented programming is a design technique that allows for the creation of complex objects by combining simpler objects. Here's a breakdown of the key points regarding composition:

Definition:

- Composition is a way to design or implement the "has-a" relationship between objects.
- It involves creating objects that contain other objects as instance variables.
- Composition represents a part-of relationship, where both entities are related, and one entity cannot exist without the other.

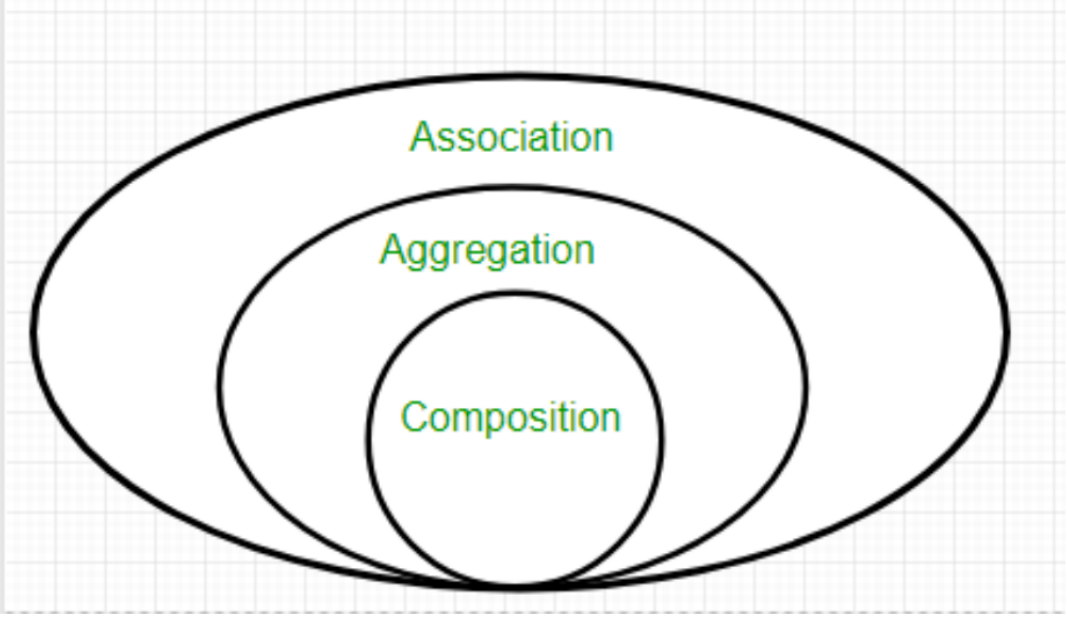
Comparison with Inheritance:

- Inheritance is used to implement the "is-a" relationship, while composition is used for the "has-a" relationship.
- Composition is favored over inheritance for code reusability and flexibility.

Benefits of Composition:

1. **Code Reusability:** Composition allows for code reuse by combining existing classes to create more complex objects.

2. **Multiple Inheritance:** In Java, composition can be used to achieve multiple inheritance by combining the functionality of multiple classes.
3. **Testability:** Composition provides better testability of a class, as each component can be tested independently.
4. **Flexibility:** It allows for easier replacement of composed class implementations with better versions, promoting flexibility and maintainability.
5. **Dynamic Behavior:** Composition enables dynamic changes to a program's behavior by changing the member objects at runtime.



In this example, composition is used to model the relationship between a `Ninja` class, which extends the `Car` class, and the `CarOil` class. The `Ninja` class has a method `NinjaOil()` that creates an instance of `CarOil` within the method, demonstrating composition.

Here's the breakdown:

```
class CarOil {
    public void FillOil() {
        System.out.println("The fuel is full in the ca
r");
    }
    public void EmptyOil() {
        System.out.println("The car has low oil");
```

```
    }
}

class Car {
    private String colour;
    private int maxi_Speed;

    public void carDetails() {
        System.out.println("Car Colour= " + colour + ",
Maximum Speed= " + maxi_Speed);
    }

    public void setColour(String colour) { // Setting co
lour of the car
        this.colour = colour;
    }

    public void setMaxiSpeed(int maxi_Speed) { // Settin
g maximum car Speed
        this.maxi_Speed = maxi_Speed;
    }
}

class Ninja extends Car {
    public void NinjaOil() {
        CarOil Ninja_Oil = new CarOil(); // Composition
        Ninja_Oil.FillOil();
    }
}

class Main {
    public static void main(String[] args) {
        Ninja NinjaCar = new Ninja();
        NinjaCar.setColour("Orange");
        NinjaCar.setMaxiSpeed(180);
        NinjaCar.carDetails();
        NinjaCar.NinjaOil();
```

```
    }
}
```

Output:

```
Car Colour= Orange, Maximum Speed= 180
The fuel is full in the car
```

In this example:

- `CarOil` and `Car` are standalone classes representing car oil-related functionality and car details, respectively.
- `Ninja` extends `Car`, indicating an "is-a" relationship where a `Ninja` is a type of `Car`.
- The `NinjaOil()` method in the `Ninja` class demonstrates composition by creating an instance of `CarOil` within the method and calling its `FillOil()` method.

So by instanstiating a instance of CarOil Ninja is getting all the methods of CarOil and by inheriting it is getting all methods of Car

- In the `Main` class, a `Ninja` object is created and its colour and maximum speed are set. The `carDetails()` method displays car details, and the `NinjaOil()` method fills the car's oil, demonstrating composition.

Example 1: Composition with Objects

```
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;

    public Car() {
```

```
        this.engine = new Engine(); // Composition
    }

    public void startCar() {
        engine.start();
        System.out.println("Car started");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.startCar();
    }
}
```

In this example, the `Car` class has a member variable `engine` which is an instance of the `Engine` class. The `Car` class uses the `Engine` class through composition.

Example 2: Composition with Interface

```
interface Engine {
    void start();
}

class ElectricEngine implements Engine {
    public void start() {
        System.out.println("Electric engine started");
    }
}

class Car {
    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine; // Composition
    }
```

```
    public void startCar() {
        engine.start();
        System.out.println("Car started");
    }
}

public class Main {
    public static void main(String[] args) {
        Engine engine = new ElectricEngine();
        Car car = new Car(engine);
        car.startCar();
    }
}
```

In this example, `Car` class has an instance variable of the `Engine` interface. The `Car` class is composed of any class implementing the `Engine` interface.

Example 3: Composition with Collections

```
import java.util.ArrayList;

class Library {
    private ArrayList<Book> books;

    public Library() {
        this.books = new ArrayList<>(); // Composition
    }

    public void addBook(Book book) {
        books.add(book);
    }

    public void displayBooks() {
        for (Book book : books) {
            System.out.println(book.getTitle());
        }
    }
}
```

```
class Book {
    private String title;

    public Book(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }
}

public class Main {
    public static void main(String[] args) {
        Library library = new Library();
        library.addBook(new Book("Book 1"));
        library.addBook(new Book("Book 2"));
        library.displayBooks();
    }
}
```

In this example, the `Library` class is composed of `Book` objects using an `ArrayList`.

OO Development Process, System Design and Frameworks

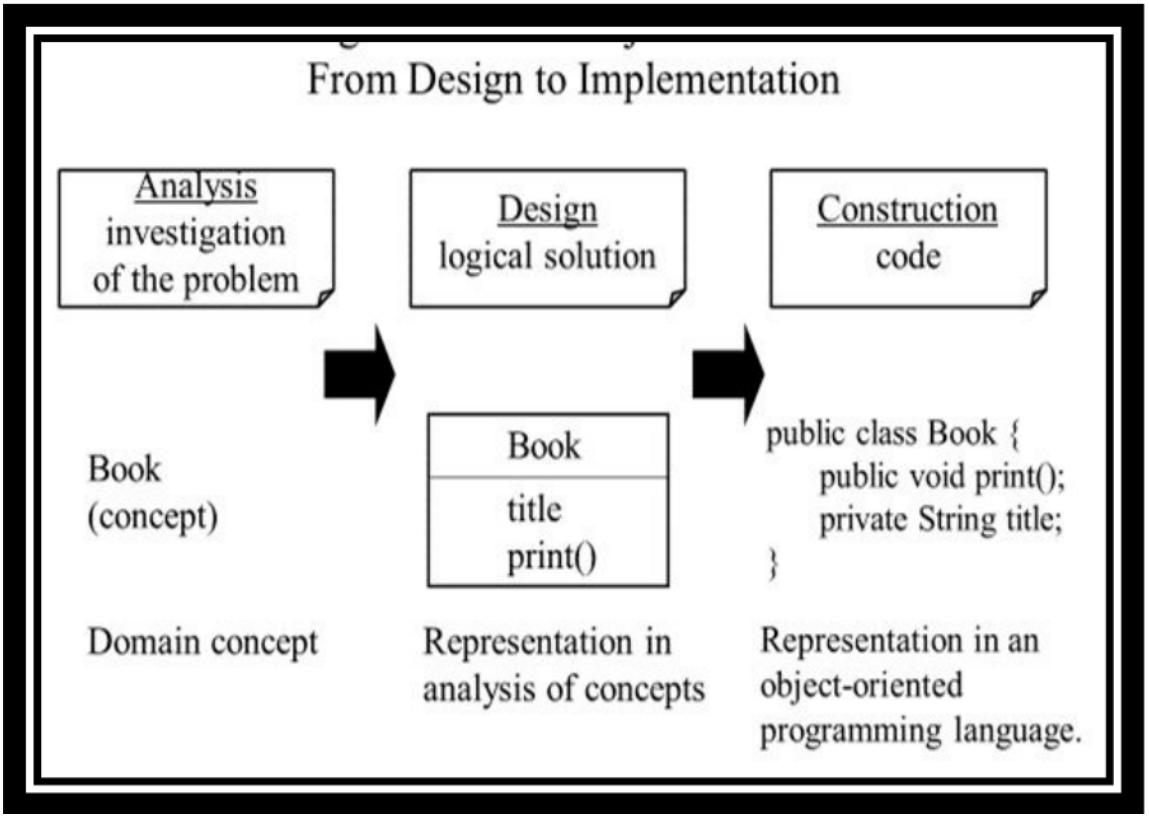
- The essence is the identification and organization of application concepts rather than their final representation in programming language.
- Encourages software developers to work and think in terms of application throughout the software life cycle.
- Encourages and facilitates re-use of software components.
- It employs international standard Unified Modeling Language (UML) from the Object Management Group (OMG).

This sequence outlines a typical software development lifecycle, particularly emphasizing Object-Oriented Methodology. Here's a breakdown of each stage:

Stages in Software Lifecycle

1. **System Conception:**
    - Begins with either business analysis or users conceiving an application and formulating tentative requirements.
2. **Analysis:**
    - Analysts work with requestors to understand the problem as problem statements are often incomplete or incorrect.
    - Develops an analysis model, which is an abstraction of what the desired system must do without implementation details.
    - Focuses on understanding the entire application, its target users, the problems it will solve, and its workflow.
3. **System Design:**
    - Development teams create a high-level strategy known as the system architecture to solve the application problem.
    - This stage establishes the overall structure and framework of the system without diving into implementation specifics.
4. **Class Design:**
    - Class designers enhance the analysis model with more details in alignment with the system design strategy.
    - Focuses on defining data structures and algorithms necessary to implement each class identified in the analysis phase.
5. **Implementation:**
    - Implementers translate the classes and relationships developed during class design into a specific programming language, database, or hardware.
    - Emphasizes following good software engineering practices to maintain traceability to the design and ensure flexibility and extensibility of the system.

Sample



System Design – Reuse Plan
• Reuse – Advantage of OO but it does not happen automatically.
• Two different aspects of reuse - Using existing things and Creating reusable things.
• Much easier to reuse existing things than to design new things.
• Most developers reuse existing things and a small fraction of developers create new things.
• Reusable things include Models, Libraries, Frameworks and Patterns.

Reuse Plan - Libraries:

- A library is a collection of classes that offer functionality useful in various contexts.
- The organization of classes within a library is crucial for ease of use.
- Each class should have accurate and comprehensive documentation to assist users in understanding its purpose and relevance.

- Good class libraries exhibit qualities such as coherence, completeness, consistency, efficiency, extensibility, and genericity.
- Libraries typically perform specific and well-defined operations, such as network protocols, compression, image manipulation, string utilities, and regular expressions.

• Several qualities of good class libraries:

Coherence: Organized about a few well focused themes;
Completeness: Provide complete behavior for the chosen theme;
Consistency: Consistent names and signatures for polymorphic operations across classes;
Efficiency: Provide alternative implementations of algorithms that trade time and space;
Extensibility: Must be able to define subclasses for library classes;
Genericity: Should use parameterized class definitions where appropriate.

Reuse Plan - Patterns:

- Patterns represent best practices or proven solutions to recurring problems in software development.
- Patterns come with guidelines on when to use them and the trade-offs involved.
- They usually involve a small number of classes and relationships.
- Advantages of patterns include the careful consideration they've received from others and their successful application to past problems.

Software Architecture Patterns: Layered Pattern, Client-Server Pattern
• Design Patterns: Gang of Four(GOF). Broken into three categories –

Creational Patterns for the creation of objects, Structural Patterns to provide relationships between objects and Behavioral Patterns tohelp define how objects interact with each other.

Reuse Plan - Frameworks:

- Frameworks provide a foundational structure or blueprint for building applications.
- They offer a skeletal structure that needs to be extended or elaborated upon to create a complete application.
- Elaboration involves specializing abstract classes within the framework to tailor the behavior to the specific requirements of the application.

- Frameworks encompass more than just classes; they also include control flow paradigms and shared conventions.
- Frameworks guide the developer in architectural decisions by providing abstract classes and predefined patterns.
- Developers customize frameworks for specific applications by subclassing and composing instances of framework classes.
- Unlike libraries where developers call functions provided by the library, frameworks control the flow of control and call the developer's code when necessary.

Library vs. Framework:

- Libraries provide collections of reusable functions or classes for specific tasks that developers call directly.
- Frameworks offer a more comprehensive structure and control flow for developing applications, with developers customizing and extending the provided framework to build their application.

Decisions - System Architecture

1. Estimating performance
2. Making a Reuse plan
3. Breaking system into sub-systems
4. Identifying Concurrency
5. Allocation of Sub Systems to hardware
6. Manage data storage
7. Handling global resources
8. Choosing a software control strategy
9. Handling boundary conditions
10. Set trade-off priorities
11. Select an architectural Style

Frameworks in Java

Frameworks play a crucial role in Java development by providing pre-built solutions and abstractions to common tasks. Here are some popular frameworks used in Java development:

1. **Collections Framework:**
    - Provides interfaces (e.g., Set, List, Queue, Deque) and implementations (e.g., ArrayList, Vector, LinkedList, PriorityQueue, HashSet,

LinkedHashSet, TreeSet) for handling collections of objects.

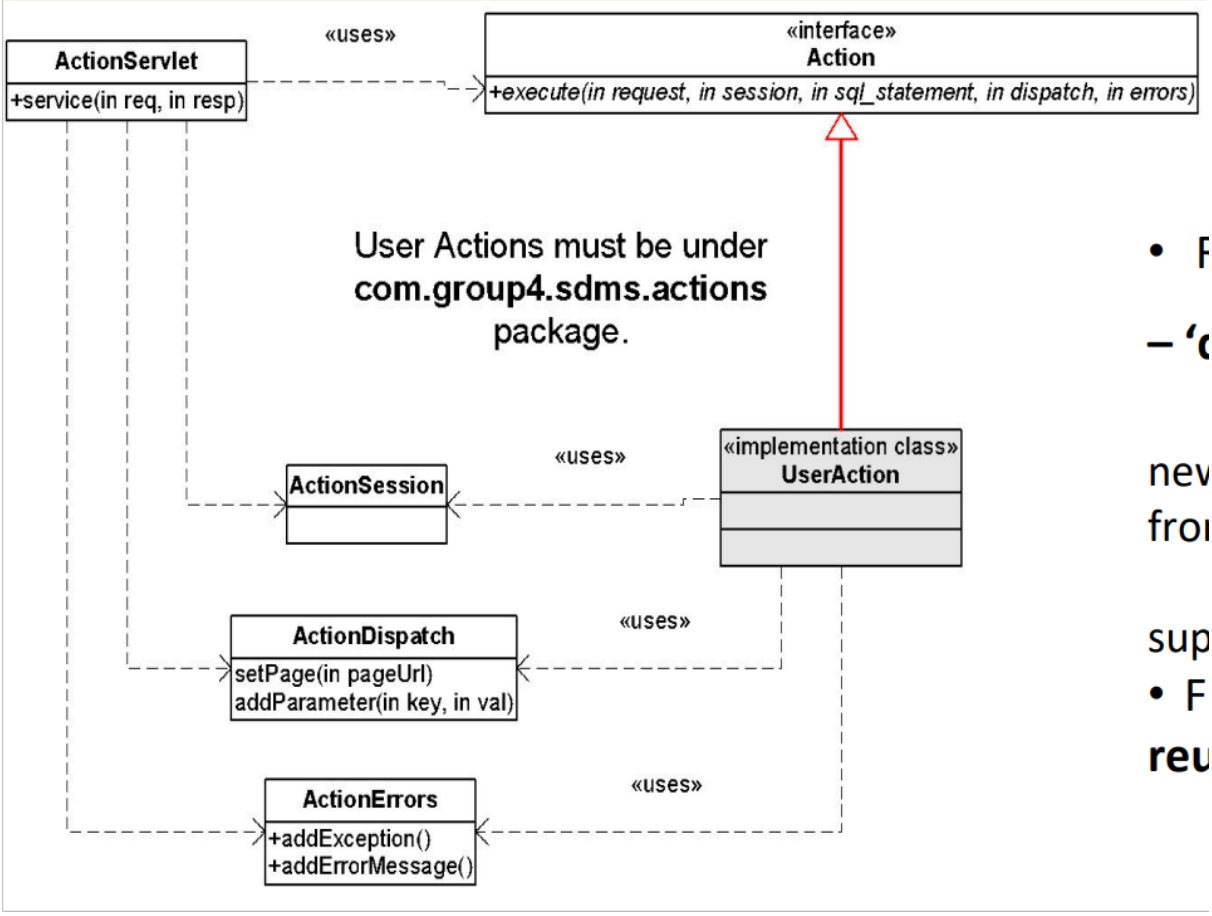
2. **Swing:**
    - Part of the Java Foundation Classes (JFC).
    - Built on top of AWT (Abstract Window Toolkit).
    - Entirely written in Java.
    - Offers a rich set of GUI components (e.g., JButton, JTextField, JCheckbox, JMenu) through the javax.swing API.
3. **AWT (Abstract Window Toolkit):**
    - Provides various component classes (e.g., Label, Button, TextField) for creating graphical user interfaces.
    - All AWT classes are part of the `java.awt` package.
4. **Spring:**
    - A comprehensive framework for building enterprise Java applications.
    - Provides features like dependency injection, aspect-oriented programming, and support for MVC architecture.
5. **Hibernate:**
    - An ORM (Object-Relational Mapping) framework for mapping Java objects to relational database tables.
    - Simplifies database operations and provides a higher level of abstraction for database interactions.
6. **Grails:**
    - A web application framework built on top of the Groovy programming language.
    - Facilitates rapid development with features like convention over configuration and seamless integration with other Java technologies.
7. **Play Framework:**
    - A lightweight, stateless web application framework for Java and Scala.
    - Emphasizes productivity, scalability, and developer-friendly features like hot reloading and asynchronous programming.
8. **JavaServer Faces (JSF):**

- A component-based MVC framework for building web applications.
- Provides reusable UI components and simplifies the development of web interfaces.

9. **Google Web Toolkit (GWT):**
    - Allows developers to write front-end web applications in Java and compile them into JavaScript.
    - Simplifies cross-browser compatibility and improves development productivity.
10. **Quarkus:**

- A Kubernetes-native Java framework optimized for fast startup times and low memory consumption.
- Ideal for building cloud-native and serverless applications.

These frameworks offer various features and abstractions that streamline Java development across different domains, including web development, enterprise applications, and GUI development. Developers choose frameworks based on project requirements, preferences, and the specific problem domains they address.



Introducing Classes

A constructor in Java is a special type of method that initializes an object when it is created. Here are some key points about constructors:

1. **Naming and Syntax:**
   - Constructors have the same name as the class they belong to.
   - They are syntactically similar to methods but do not have an explicit return type, not even `void`.
2. **Initialization:**
   - Constructors are typically used to initialize instance variables of the class or perform any other necessary startup procedures to create a fully formed object.
3. **Default Constructor:**

- If you do not explicitly define any constructors in a class, Java provides a default constructor automatically.
- This default constructor initializes all member variables to zero or corresponding default values.
- However, once you define your own constructor(s), the default constructor is no longer added automatically.

4. **Invocation:**
    - Constructors are invoked each time an object is created using the `new` operator.
    - When you create an object, the constructor is called to assign initial values to the data members of the class.

Example of a constructor in Java:

```
public class MyClass {
    private int myVar;

    // Constructor
    public MyClass() {
        // Initialize instance variables
        myVar = 0;
        // Other startup procedures can be performed here
    }

    // Other methods and variables can be defined here
}
```

In the above example, `MyClass` has a constructor that initializes the `myVar` instance variable to zero when an object of `MyClass` is created.

1. **Definition and Characteristics:**
    - A constructor initializes an object when it is created and shares the same name as its class.
    - Constructors resemble methods but have no explicit return type.
    - They are typically used to initialize instance variables or perform startup procedures for creating a fully formed object.

- All classes have constructors, and if one isn't explicitly defined, Java automatically provides a default constructor.

2. **Default Constructor:**
   - A constructor with no parameters.
   - Provided by the compiler if no constructor is defined in the class.
   - Initializes object fields with default values (e.g., 0, false, null).
3. **Parameterized Constructor:**
   - A constructor with parameters used to initialize object fields with given values.
   - No return value statements in constructors, but they return the current class instance.

```
// Java Program for Parameterized Constructor
import java.io.*;
class Geek {
// data members of the class.
String name;
int id;
Geek(String name, int id)
{
this.name = name;
this.id = id;
}
}
class GFG {
public static void main(String[] args)
{
// This would invoke the parameterized constructor.
Geek geek1 = new Geek("avinash", 68);
System.out.println("GeekName :" + geek1.name
+ " and GeekId :" + geek1.id);
}
}
```

College Examples

```
class Box
{
    double width;
    double height;
    double depth;

    Box()
    {
        width=0;
        height=0;
        depth=0;
    }

    Box(int w,int h,int d)//constructor can take parameter
    {
        width=w;
        height=h;
        depth=d;
    }
    Box(Box b1)//copying from existing obj
    {
        width=b1.width;
        height=b1.height;
        depth=b1.depth;
    }
    void disp()
    {
        System.out.println("width: "+width);
        System.out.println("height: "+height);
        System.out.println("depth: "+depth);
    }

}

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

```
        Box b1=new Box();
        Box b2=new Box(10,20,30);
        Box b3=new Box(b2);
        System.out.println("-------object b1 details------
        b1.disp();
        System.out.println("-------object b2 details------
        b2.disp();
        System.out.println("-------object b3 details------
        b3.disp();
    }
}
```

```
class rect
{
 int l;//instance variables
 int b;


rect()
{
this(10,20);//parametrized ctt
l=b=0;
System.out.println("Default ctt");
}

rect(int l, int b)
{
this.l=l;//instance var are hidden by local variables.
this.b=b;
System.out.println("param ctt");
}

rect(rect t)
{
this.l=t.l;
this.b=t.b;
System.out.println("ctt with object as parameter");
}
```

```
}


class CttDemo1

{
public static void main(String args[])

{

rect r = new rect();
rect r1=new rect(10,20);

rect r2=new rect(r1);

System.out.println("Object info");

System.out.println(r.l);
System.out.println(r.b);

System.out.println(r1.l);
System.out.println(r1.b);

System.out.println(r2.l);
System.out.println(r2.b);

}
}
```

- When `r2` is created using `rect r2 = new rect(r1);`, it invokes the constructor `rect(rect t)`, where `t` is the `r1` object.
- Inside the constructor `rect(rect t)`, `this.l = t.l;` and `this.b = t.b;` copies the values of `l` and `b` from the `t` object (`r1`) into the `r2` object being created.

So, `r2` has its own memory space, but its `l` and `b` values are initialized with the values of `l` and `b` from `r1`.

4. **Copy Constructor:**

- Used to create an exact copy of an existing object.

```
// Java Program for Copy Constructor
import [java.io](http://java.io/).*;

class Geek {
// data members of the class.
String name;
int id;// Parameterized Constructor
Geek(String name, int id)
{
    this.name = name;
    this.id = id;
}

// Copy Constructor
Geek(Geek obj2)
{
    this.name = obj2.name;
    this.id = obj2.id;
}
}
class GFG {
public static void main(String[] args)
{
// This would invoke the parameterized constructor.
System.out.println("First Object");
Geek geek1 = new Geek("avinash", 68);
System.out.println("GeekName :" + [geek1.name](http://ge
ek1.name/)
+ " and GeekId :" + [geek1.id](http://geek1.id/));
    System.out.println();

    // This would invoke the copy constructor.
    Geek geek2 = new Geek(geek1);
    System.out.println(
        "Copy Constructor used Second Object");
```

```
    System.out.println("GeekName :" + geek2.name
                       + " and GeekId :" + geek2.id);
}
}
```

5. **Access Modifiers:**
    - Constructors can have access modifiers like private, public, protected, or default.
    - If a constructor is made private, objects of that class cannot be created from outside the class.

```
class A {
    private A() {} // Private constructor

    void msg() {
        System.out.println("Welcome to OOAD with Java clas
s");
    }
}

public class Sample {
    public static void main(String args[]) {
        A obj = new A(); // Compile Time Error due to priva
te constructor
    }
}
```

In this example, class `A` has a private constructor, preventing the creation of objects of class `A` from outside the class. Consequently, attempting to create an object of class `A` in the `Sample` class results in a compile-time error.

How Java Constructors are Different From Java Methods?

- Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or **void** if does not return any value.

- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

In Java, when you define a constructor for a class that extends another class (a subclass), the first line of the constructor typically calls either `super()` or `this()`.

Here's what each of these calls means:

1. `super()`:
    - Calls the constructor of the superclass (the parent class).
    - This ensures that the superclass's constructor is executed before the subclass's constructor.
    - If you don't explicitly call `super()` in your subclass constructor, the compiler will automatically insert a call to the superclass's default (no-argument) constructor.
2. `this()`:
    - Calls another constructor within the same class (constructor chaining).
    - It can be used to avoid duplicating initialization code across multiple constructors in the same class.
    - Like `super()`, if you don't explicitly call `this()` in your constructor, the compiler will not insert it automatically.

Here's an example demonstrating the use of `super()`:

```
class Parent {
    Parent() {
        System.out.println("Parent constructor");
    }
}

class Child extends Parent {
    Child() {
        // Implicit call to super() is inserted here by the
compiler
        System.out.println("Child constructor");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Child child = new Child();
    }
}
```

In the above example, when an object of `Child` class is created, the constructor of the superclass `Parent` is automatically called first due to the implicit `super()` call inserted by the compiler. Then, the constructor of the `Child` class is executed.

The provided code defines two classes: `Parent` and `Child`. `Child` extends `Parent`. When an object of `Child` is created, the constructor of `Child` is called first. If `Child` has no explicit call to `super()` (i.e., the constructor of the parent class), Java automatically inserts a call to the no-argument constructor of the parent class.

Here's the sequence of events:

1. An object of `Child` class is created in the `main` method.
2. The constructor of `Child` is called.
3. Inside the `Child` constructor, there's an implicit call to `super()` which invokes the constructor of the `Parent` class.
4. The constructor of `Parent` is executed first, printing `"Parent constructor"`.
5. After the execution of the parent constructor, the execution returns to the `Child` constructor and continues, printing `"Child constructor"`.

So, the output of running the provided code will be:

```
Parent constructor
Child constructor
```

More Examples on super()

```
class base {
    static int i = 100;
}
```

```
class child extends base {
    int i = 200;

    void display() {
        System.out.println(super.i); // 100
        System.out.println(i); // 200
    }
}

class superdemo {
    public static void main(String args[]) {
        child cobj = new child();
        cobj.display();
    }
}
```

```
class base {
    void display() {
        System.out.println("base class display");
    }
}

class child extends base {
    void display() {
        System.out.println("child class display");
    }

    void print() {
        super.display();
    }
}

class superdemo1 {
    public static void main(String args[]) {
        child cobj = new child();
        cobj.display(); // child class display
        cobj.print();   // base class display
```

```
    }
}
```

```
class base {
    int i;

    base() {
    }

    base(int i) {
        System.out.println("base class param ctt");
        this.i = i;
    }
}

class child extends base {
    int j;

    child(int i, int j) {
        super(i);
        this.j = j;
        System.out.println("child class param ctt");
    }
}

class superdemo2 {
    public static void main(String args[]) {
        child cobj = new child(10, 20);
    }
}
```

```
base class param ctt
child class param ctt
```

Garbage Collector

Garbage Collection (GC):

- Java Garbage Collection is a process that identifies and removes unused objects from memory to free up space.
- Unlike languages like C, where memory allocation and deallocation are manual processes, Java automatically manages memory using Garbage Collection.

Garbage Collector:

- The Garbage Collector is a program running in the background that examines all objects in memory to identify those not referenced by any part of the program.
- It removes these unreferenced objects, reclaiming space for allocation to other objects.

Actions Before Object Destruction:

- Certain actions need to be performed before an object is destroyed to ensure proper cleanup and resource release.
- Examples include closing database connections or files, releasing network resources, performing housekeeping tasks, and releasing heap space allocated during the object's lifetime.

Finalization Mechanism:

- Java provides a mechanism called finalization to perform cleanup actions before an object is destroyed.
- This mechanism involves using the `finalize()` method, which is called by the Garbage Collector before reclaiming the memory occupied by an object.

The `finalize()` method in Java serves as a hook that allows you to perform cleanup actions before an object is garbage-collected. Here's a breakdown of the general form and behavior of the `finalize()` method:

```
protected void finalize() {
    // Finalization code here
    // Specify actions that must be performed before an obj
ect is destroyed
}
```

- **Syntax**: The `finalize()` method is declared with the `protected` access modifier and has a return type of `void`. It doesn't accept any parameters.
- **Functionality**: Within the method body, you can specify the finalization code, which typically includes actions necessary for cleanup before the object is reclaimed by the garbage collector.
- **Invocation**: Java runtime automatically invokes the `finalize()` method on an object just before it is garbage-collected. This happens when there are no more references to the object, and the garbage collector identifies it as eligible for collection.
- **Access Modifier: The `protected` access modifier is used to restrict access to the `finalize()` method. It prevents the method from being accessed by code defined outside the class hierarchy. This ensures that finalization actions are controlled by the class itself.**
- **Note**: It's important to understand that the `finalize()` method is not called when an object goes out of scope or when it is explicitly dereferenced. Instead, it is invoked specifically prior to garbage collection, providing an opportunity for resource cleanup and other necessary actions.

Parameter Passing – Value Types and Reference Types

Types of Parameters

- Formal Parameter
- Actual Parameter

Parameter passing techniques

- Pass by Value
- Pass by Reference

1. Formal Parameter and Actual Parameter:

In a method or function declaration, formal parameters are the parameters defined in the function signature, while actual parameters are the arguments passed when calling the function.

```
public class ParameterExample {
    // Method with formal parameter "name"
    public static void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }

    public static void main(String[] args) {
        String userName = "Alice"; // Actual parameter
        greet(userName); // Calling the method with actual
parameter
    }
}
```

In the above example:

- `name` is the formal parameter defined in the `greet` method.
- `userName` is the actual parameter passed when calling the `greet` method.

2. Parameter Passing Techniques:

a. Pass by Value:

In pass by value, a copy of the value of the actual parameter is passed to the formal parameter.

```
public class PassByValueExample {
    public static void increment(int num) { //takes only co
py of Actual Parameter becuase you are referencing datatype
        num++; // Incrementing the formal parameter
        System.out.println("Inside method: " + num);
    }

    public static void main(String[] args) {
        int number = 5; // Actual parameter
        increment(number); // Calling the method with actua
l parameter
        System.out.println("Outside method: " + number);
```

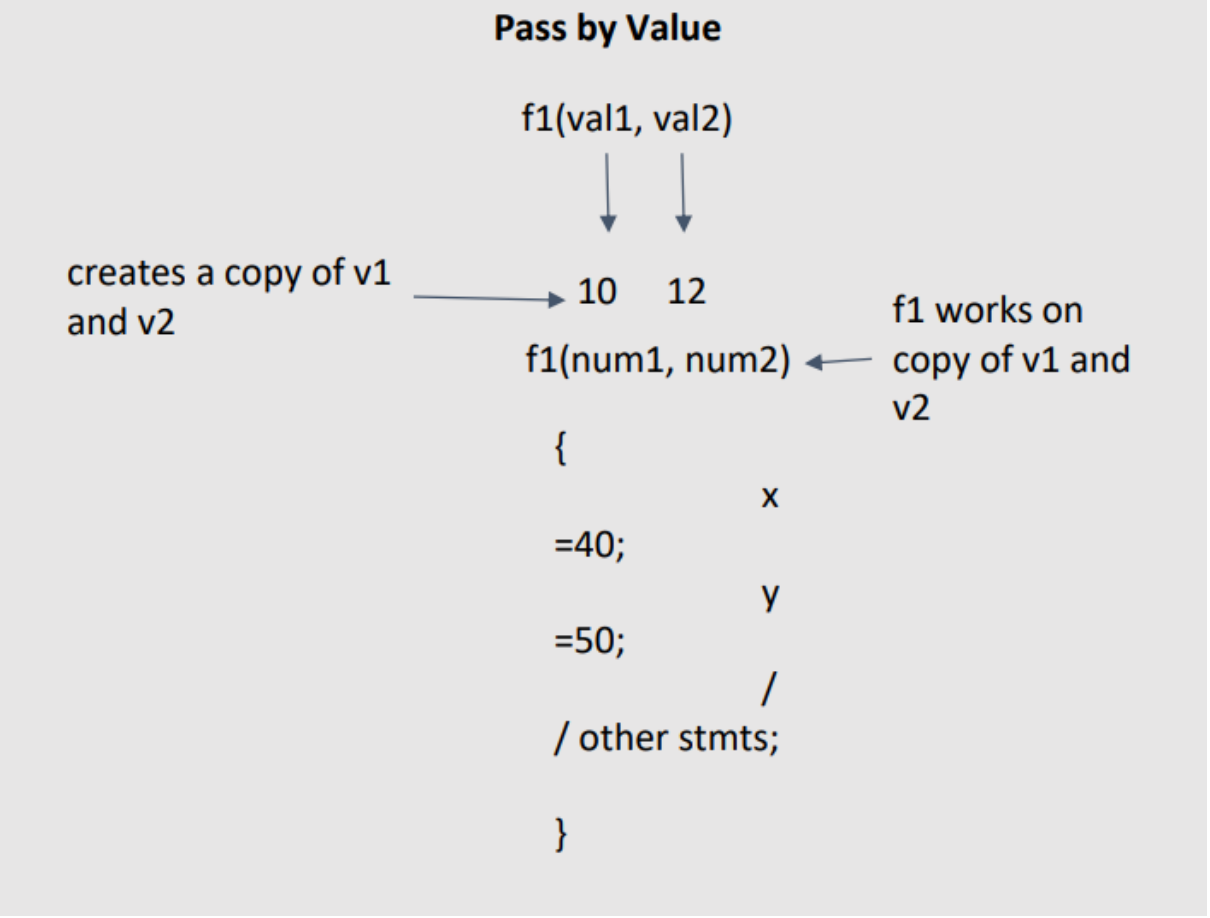
```
    }
}
```

Output:

```
Inside method: 6
Outside method: 5
```

In the above example, the value of `number` remains unchanged outside the `increment` method because Java passes primitive types (like `int`) by value, meaning changes to the formal parameter `num` inside the method do not affect the actual parameter `number`.

```
class callbyvaluedemo
{
    public static void main(String[] args)
    {   int data = 7;
        parameter p1 = new parameter();
        System.out.println(data);//7
        p1.testing(data);
        System.out.println(data);//7
    }
}
class parameter{
    int testing(int data)
    {       data = 89;
            return data;    }
}
```



b. Pass by Reference:

In pass by reference, a reference to the memory location of the actual parameter is passed to the formal parameter.

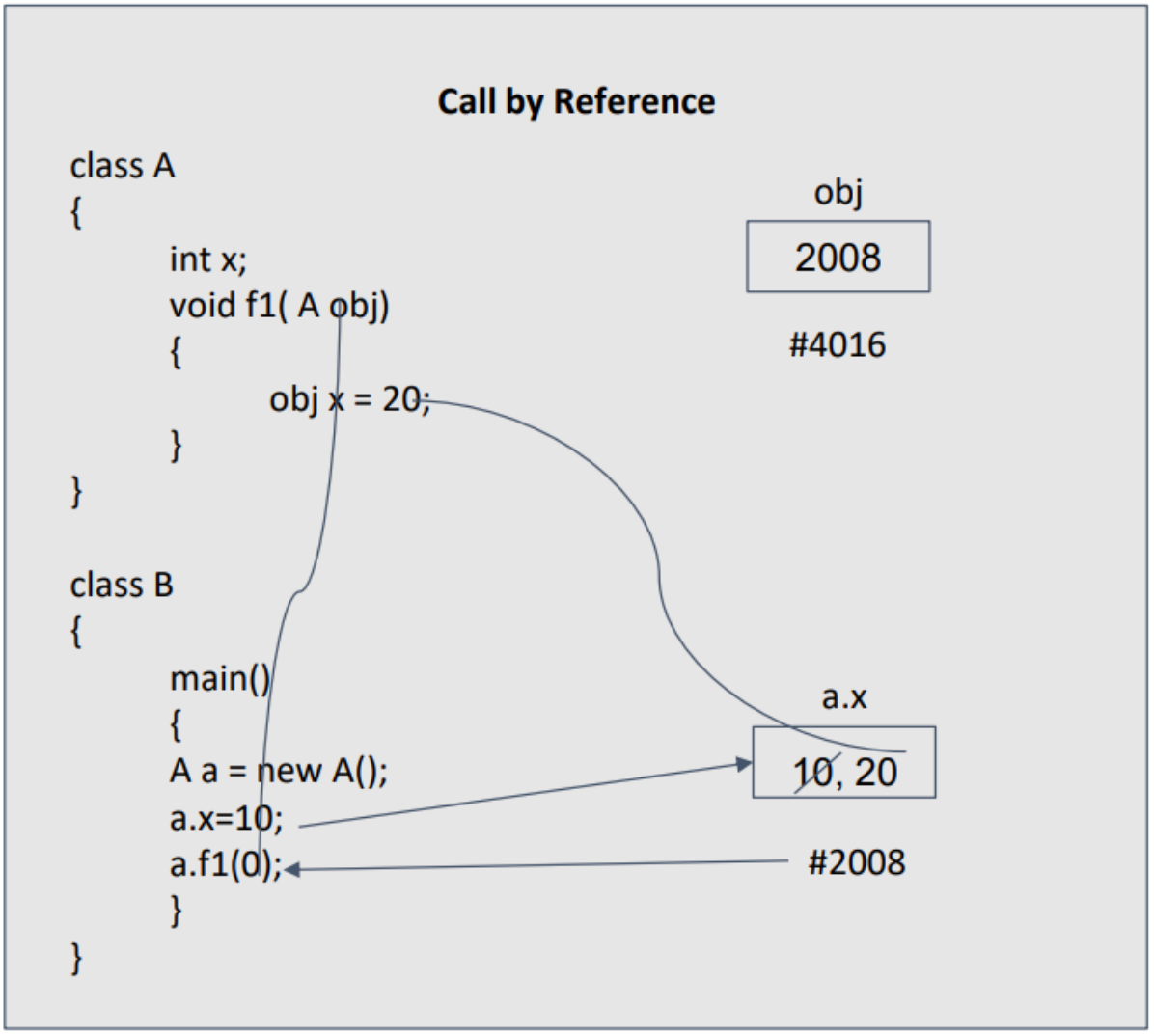
```
public class PassByReferenceExample {
    public static void append(StringBuilder str) {//referen
cing the object not data type
        str.append(" World!"); // Modifying the formal para
meter
        System.out.println("Inside method: " + str);
    }

    public static void main(String[] args) {
        StringBuilder message = new StringBuilder("Hello");
// Actual parameter
        append(message); // Calling the method with actual
parameter
        System.out.println("Outside method: " + message);
```

```
    }
}
```

Output:

```
Inside method: Hello World!
Outside method: Hello World!
```

In the above example, changes made to the `StringBuilder` object `str` inside the `append` method are reflected in the `message` object outside the method. Java passes objects (like `StringBuilder`) by reference, meaning modifications to the formal parameter inside the method affect the actual parameter.



1. **Changes Not Transmitted Back to Caller:**

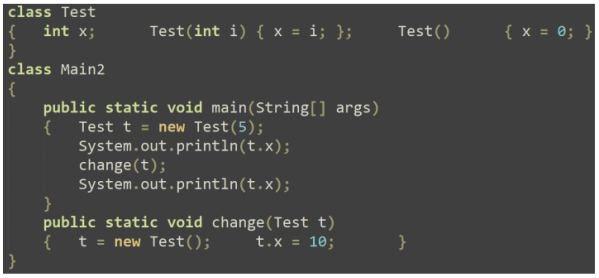
- When passing arguments by value, any changes made to the formal parameter (the parameter inside the function or method) do not affect the actual parameter (the argument passed by the caller).
- This means that modifications to the formal parameter variable inside the called function or method only affect the separate storage location allocated for that parameter within the function or method, and these changes are not reflected in the actual parameter in the calling environment.

2. **Separate Storage Location:**
   - Each parameter passed by value typically has its own separate storage location within the called function or method.
   - Any modifications made to the value of the parameter variable inside the function or method are made to this separate storage location and do not affect the original value of the actual parameter passed by the caller.
3. **Non-Primitive Types are References:**
   - Non-primitive types, also known as reference types, include objects, arrays, and other complex data structures.
   - When passing a non-primitive type as a parameter to a function or method, what is actually passed is a reference (or pointer) to the object in memory rather than the object itself.
4. **Changes Transmitted Back to Caller:**
   - With reference types, changes made to the formal parameter (the parameter inside the function or method) are transmitted back to the caller through parameter passing.
   - This means that any modifications to the formal parameter variable inside the called function or method affect the same object referenced by the actual parameter in the calling environment.
5. **Reflection of Changes:**
   - Changes made to the formal parameter are reflected in the actual parameter in the calling environment because both the formal and actual parameters reference the same object in memory.

- As a result, any modifications made to the object through the formal parameter inside the function or method will be visible to the caller after the function or method returns.

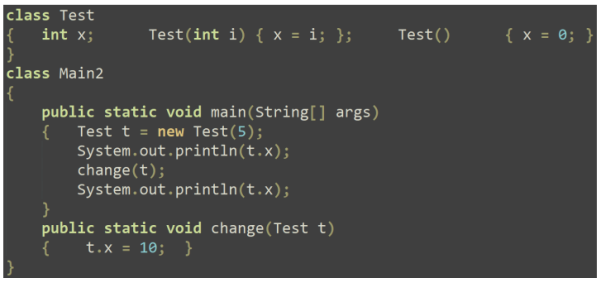
Reference types continued..

- **Trick 1: The changes are not reflected back if we change the object itself to refer some other location or object**



```
class Test
{   int x;      Test(int i) { x = i; };     Test()      { x = 0; }
}
class Main2
{
    public static void main(String[] args)
    {   Test t = new Test(5);
        System.out.println(t.x);
        change(t);
        System.out.println(t.x);
    }
    public static void change(Test t)
    {   t = new Test();     t.x = 10;       }
}
```

- **Trick 2: Changes are reflected back if we do not assign reference to a new location or object**



```
class Test
{   int x;      Test(int i) { x = i; };     Test()      { x = 0; }
}
class Main2
{
    public static void main(String[] args)
    {   Test t = new Test(5);
        System.out.println(t.x);
        change(t);
        System.out.println(t.x);
    }
    public static void change(Test t)
    {    t.x = 10;  }
}
```

```
class ParamDemo {
    public static void main(String a[]) {
        Test t = new Test(10, 20);
        t.display();
        t.testMethod(t);
        t.display();
    }
}

class Test {
    int i;
    int j;

    Test(int i, int j) {
        this.i = i;
        this.j = j;
    }

    Test(Test p) {
        this.i = p.i;
        this.j = p.j;
```

```
    }

    void testMethod(Test t1) {
        Test t2 = new Test(t1);
        t2.i++;
        t2.j++;
    }

    void display() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

```
i=10 j=20
i=10 j=20
```

Overloading of Methods (Dynamic Method dispatch)

Method overloading is a fundamental feature in object-oriented programming languages like Java, allowing a class to have multiple methods with the same name but different parameter lists. Here's an explanation of key points about method overloading:

1. **Definition and Characteristics:**
    a. Method overloading has no relation to return type
    - Method overloading enables a class to define multiple methods with the same name but different parameter lists.
    - This allows programmers to create methods that perform similar tasks but operate on different types of data or accept different numbers or types of parameters.
2. **Compile-Time Polymorphism:**
    - **Method overloading is a form of compile-time polymorphism, also known as static polymorphism or early binding.**
    - The decision about which overloaded method to call is made by the compiler based on the arguments passed at compile time.

3. **Ways to Overload Methods:**
    - There are three ways to overload methods:
        - Changing the number of parameters
        - Changing the data type of parameters
        - Changing the order of parameters of methods
4. **Benefits of Method Overloading:**
    - Increases the readability of the program by providing meaningful method names for different variations of functionality.
    - Provides flexibility to programmers by allowing them to call the same method with different types or numbers of parameters.
    - Improves code cleanliness by reducing the need for multiple method names that perform similar tasks.
    - Reduces execution time because method binding is done at compile time, avoiding the need for runtime method resolution.
    - Minimizes code complexity by consolidating similar functionality into a single method name.
    - Facilitates code reusability, as overloaded methods can be used in various contexts, saving memory and promoting modular design.

```
class Rect {
    int l, b;

    void add() {
        System.out.println("output=  0");
        System.out.println("You are in zero-argument method")
    }

    void add(int a) {
        System.out.println("output=  " + a);
        System.out.println("You are in one-argument method of
    }

    void add(int a, int b) {
        System.out.println("output=  " + (a + b));
```

```
        System.out.println("You are in two-argument method of
    }

    void add(int a, double b) {
        System.out.println("output=  " + (a + b));
        System.out.println("You are in two-argument method of
    }

    void add(double a, int b) {
        System.out.println("output=  " + (a + b));
        System.out.println("You are in two-argument method of
    }

    void add(double a, double b) {
        System.out.println("output=  " + (a + b));
        System.out.println("You are in two-argument method of
    }
}

public class Demo {
    public static void main(String args[]) {
        Rect r = new Rect();

        r.add();
        r.add(1);
        r.add(1, 2);
        r.add(1, 2.4);
        r.add(2.3, 1);
        r.add(1.1, 3.5);
    }
}
```

Methods

1. Instance Methods:

- Instance methods are methods that require an object of their class to be created before they can be called.

- To invoke an instance method, you need to create an object of the class in which the method is defined.
- Instance methods are associated with objects of the class and can access instance variables and other instance methods.

2. Static Methods:

- Static methods are methods in Java that can be called without creating an object of the class.
- They are referenced by the class name itself or by a reference to the object of that class.
- Static methods are not associated with any specific instance of the class and cannot access instance variables or instance methods directly.

Example Illustrating Instance Methods:

```
class Sample {
    String name = ""; // Instance variable

    // Instance method to set the name
    public void setName(String s) {
        this.name = s;
    }
}

class Example1 {
    public static void main(String[] args) {
        // Create an instance of the class
        Sample obj1 = new Sample();

        // Calling an instance method to set the name
        obj1.setName("Ramu");

        // Accessing instance variable directly
        System.out.println(obj1.name); // Output: Ramu
    }
}
```

In the above example:

- `Sample` class has an instance method `setName` that sets the value of the `name` instance variable.
- In the `main` method of the `Example1` class, an object `obj1` of the `Sample` class is created.
- The `setName` instance method is called on `obj1` to set the name to "Ramu".
- The instance variable `name` is accessed directly and its value is printed, which results in printing "Ramu".

```
class Counter{
    int count = 0; // instance variable

    Counter(){
        count++; // incrementing value
        System.out.println(count);
    }

    public static void main(String args[]){
        // Creating objects
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter c3 = new Counter();
    }
}
```

```
1
1
1
```

Static Method

1. Static Variables (Class Variables):

- Static variables, also known as class variables, are common to all objects of the class.
- They are associated with the class itself rather than with individual objects.

- Every instance of the class shares the same static variable, which is stored in a fixed location in memory.
- Static variables can be accessed and modified by any object of the class, or even without creating an instance of the class.

Example:

```
class Student{
    int rollno; //instance variable
    String name;
    static String college = "PESU"; //static variable

    //constructor
    Student(int r, String n){
        rollno = r;
        name = n;
    }

    //method to display the values
    void display (){
        System.out.println(rollno + " " + name + " " + coll
ege);
    }
}

//Test class to show the values of objects
public class TestStaticVariable{
    public static void main(String args[]){
        Student s1 = new Student(123, "Ramu");
        Student s2 = new Student(456, "Abhay");

        // we can change the college of all objects by the
single line of code
        Student.college = "PES University";

        s1.display();
        s2.display();
```

```
    }
}
```

For static you dont have to create an instance of the class

```
123 Ramu PES University
456 Abhay PES University
```

When you use static variable all instances having the static variable can be changed at once

```
class Counter2{
    static int count = 0; // static variable

    Counter2(){
        count++; // incrementing the value of static variable
        System.out.println(count);
    }

    public static void main(String args[]){
        // creating objects
        Counter2 c1 = new Counter2();
        Counter2 c2 = new Counter2();
        Counter2 c3 = new Counter2();
    }
}
```

```
1
2
3
```

2. Static Methods (Class Methods):

- Static methods belong to the class rather than to any specific instance.
- They can be called without creating an object of the class and are accessed using the class name.

- Static methods cannot access instance variables or instance methods directly, as they are not associated with any particular object.

```
class Sample {
    public static String name = "";
    public static void setname(String s) {
        name = s;
    }
}

class Example2 {
    public static void main(String[] args) {
        // Accessing the static method setname
        // and field by class name itself.
        Sample.setname("abhiram");
        System.out.println(Sample.name);

        // Accessing the static method setname
        Sample obj = new Sample();
        obj.setname("manish");
        System.out.println(obj.name);
    }
}
```

```
abhiram
manish
```

3. Static Blocks:

- Static blocks are used to initialize static data members.
- They are executed before the `main` method, at the time of class loading, and are executed only once.
- Static blocks are typically used for static variable initialization or performing other one-time initialization tasks.

```
public class StaticDemo {
    static int a = 1, b; // static variables
```

```
    static void display(int x) {
        System.out.println("x=" + x); // prints the value of 
        System.out.println("a=" + a); // prints the value of 
        System.out.println("b=" + b); // prints the value of 
    }

    static { // static block
        System.out.println("Inside static block");
        b = a * 2; // initialize the value of b in static blo
    }

    public static void main(String[] args) {
        System.out.println("Inside main");
        display(5); // calling display method
    }
}
```

```
Inside static block
Inside main
x=5
a=1
b=2
```

In Java, static blocks are executed when the class is loaded into memory, before the execution of the main method. Here's the order of execution:

1. **Static Block Execution**:
    - Static blocks are executed when the class is loaded into memory, before any object of the class is created.
    - Static variables are initialized inside static blocks.
2. **Main Method Execution**:
    - The `main` method is executed after the static blocks have been executed.

In your code:

```
static int a = 1, b; // static variables

static { // static block
    System.out.println("Inside static block");
    b = a * 2; // initialize the value of b in static block
}
```

The static block is executed first, before the `main` method, and `b` is initialized to `a * 2`, which is `1 * 2 = 2`.

So, when you run the program, you will see the output:

```
Inside static block
Inside main
x=5
a=1
b=2
```

4. Static Nested Classes:

- A nested class can be made static, known as a static nested class.
- Static nested classes do not require an instance of the outer class to be instantiated.
- They cannot access non-static members of the outer class, as they are not associated with any particular instance.

Inheritance:

- **Definition**: Inheritance is a fundamental feature of object-oriented programming (OOP) where one class (subclass) can inherit the properties (fields and methods) of another class (superclass).
- **IS-A Relationship**: Inheritance represents an IS-A relationship, also known as a parent-child relationship, where a subclass is a specific type of the superclass.
- **Code Reusability**: Inheritance enables code reusability by allowing subclasses to inherit and reuse the features defined in the superclass.

- **Superclass and Subclass**:
  - The class whose features are inherited is called the superclass, base class, or parent class.
  - The class that inherits the features from another class is called the subclass or child class.

Example of Inheritance in Java:

```
class Teacher {
    String designation = "Teacher";
    String collegeName = "PESU";

    void does() {
        System.out.println("Teaching");
    }
}

public class JavaTeacher extends Teacher {
    String mainSubject = "Java";

    public static void main(String args[]) {
        JavaTeacher obj = new JavaTeacher();
        System.out.println(obj.collegeName); // Output: PES
U
        System.out.println(obj.designation); // Output: Tea
cher
        System.out.println(obj.mainSubject); // Output: Jav
a
        obj.does(); // Output: Teaching
    }
}
```

In this example:

- `Teacher` class is the superclass with properties `designation` and `collegeName`, and method `does()`.
- `JavaTeacher` class is the subclass of `Teacher`, inheriting its properties and method.

- `JavaTeacher` class adds its own property `mainSubject`.
- In the `main` method, an object of `JavaTeacher` is created and properties and method inherited from `Teacher` class are accessed.

Limitation of Inheritance:

- **Single Inheritance**: In Java, a class cannot inherit from more than one superclass simultaneously. Java supports single inheritance, meaning a subclass can only inherit from one superclass. This limitation helps in avoiding ambiguity and complexity in the inheritance hierarchy.

Overall, inheritance is a powerful mechanism in Java that promotes code reusability, modularity, and hierarchy in object-oriented programming. It allows for building complex class structures with shared functionality and specialization.

Advantages of Inheritence

1. **Readability Improvement**:
    - Inheritance enhances code readability by promoting a hierarchical structure that reflects real-world relationships between classes.
    - By inheriting properties and methods from a superclass, subclasses can focus on their unique functionalities, making the code easier to understand and maintain.
    - Developers can easily comprehend the codebase by leveraging inheritance, as it provides a clear and organized way to represent class relationships.
2. **Time Saving**:
    - Inheritance saves time by allowing developers to reuse existing code from a superclass rather than reinventing the wheel.
    - Instead of writing redundant code for common functionalities, developers can extend existing classes and build upon their functionality.
    - This approach minimizes the need for duplicating code, leading to faster development cycles and reduced time-to-market for software projects.
3. **Method Overriding**:

- Inheritance enables method overriding, a powerful feature that allows subclasses to provide their own implementation of methods inherited from the superclass.
- By overriding methods, developers can tailor the behavior of the subclass to suit specific requirements or use cases.
- This flexibility ensures that each subclass can have its own meaningful implementation of methods inherited from the superclass, enhancing the extensibility and adaptability of the codebase.

Types of Inheritence

1. Single Level Inheritance:

```
class Employee {
    float salary = 40000;
}

class Programmer extends Employee {
    int bonus = 10000;

    public static void main(String args[]) {
        Programmer p = new Programmer();
        System.out.println("Programmer salary is: " + p.sal
ary);
        System.out.println("Bonus of Programmer is: " + p.b
onus);
    }
}
```

In single-level inheritance, the `Programmer` class inherits from the `Employee` class. The `Programmer` class inherits the `salary` variable from the `Employee` class.

2. Multilevel Inheritance:

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

```
}

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

class BabyDog extends Dog {
    void weep() {
        System.out.println("weeping...");
    }
}

class TestInheritance2 {
    public static void main(String args[]) {
        BabyDog d = new BabyDog();
        d.weep();
        d.bark();
        d.eat();
    }
}
```

In multilevel inheritance, the `BabyDog` class inherits from the `Dog` class, which in turn inherits from the `Animal` class. The `BabyDog` class inherits the `eat()` method from the `Animal` class.

3. Hierarchical Inheritance:

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

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

```
}

class Cat extends Animal {
    void meow() {
        System.out.println("meowing...");
    }
}

class TestInheritance3 {
    public static void main(String args[]) {
        Cat c = new Cat();
        c.meow();
        c.eat();
        // c.bark(); // This line will cause a compile-time
error
    }
}
```

In hierarchical inheritance, both the `Dog` and `Cat` classes inherit from the `Animal` class. They share the `eat()` method from the `Animal` class, but each has its own unique method (`bark()` for `Dog` and `meow()` for `Cat`).

Method Overriding

Definition:

- **Method Overriding**: When a subclass (child class) provides a specific implementation of a method that is already declared in its parent class, it is known as method overriding.
- Method overriding is essential for achieving runtime polymorphism in Java.

Usage:

- **Providing Specific Implementation**: Method overriding is used to provide a specific implementation of a method defined in the superclass.
- **Runtime Polymorphism**: Method overriding enables runtime polymorphism, where the actual method to be executed is determined at runtime based on the type of object.

Rules for Method Overriding:

1. The method must have the same name as in the parent class.
2. The method must have the same parameters as in the parent class.
3. There must be an IS-A relationship between the superclass and subclass (inheritance).

Example:

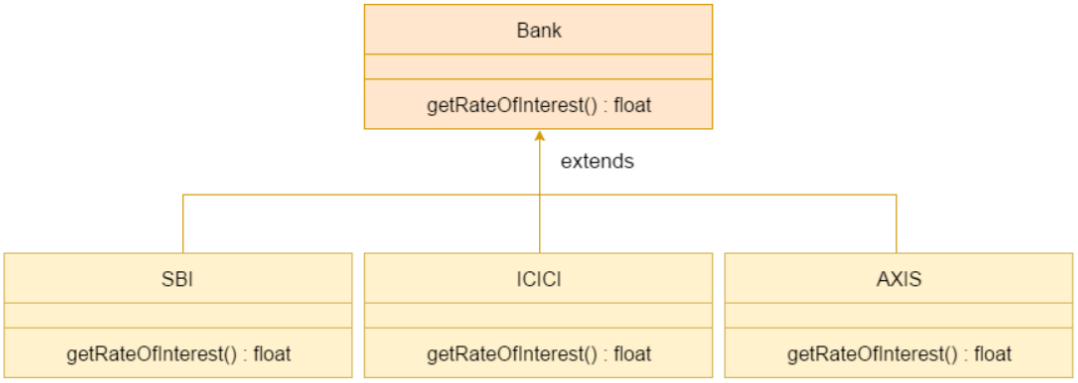
Consider a scenario where different banks provide different interest rates. We can use method overriding to implement this behavior.

```
class Bank {
    float getRateOfInterest() {
        return 0;
    }
}

class SBI extends Bank {
    float getRateOfInterest() {
        return 8.0f; // SBI offers 8% interest rate
    }
}

class ICICI extends Bank {
    float getRateOfInterest() {
        return 7.0f; // ICICI offers 7% interest rate
    }
}

class AXIS extends Bank {
    float getRateOfInterest() {
        return 9.0f; // AXIS offers 9% interest rate
    }
}

public class TestBank {
    public static void main(String args[]) {
        SBI sbi = new SBI();
```

```
        ICICI icici = new ICICI();
        AXIS axis = new AXIS();
        System.out.println("SBI Interest Rate: " + sbi.getR
ateOfInterest());
        System.out.println("ICICI Interest Rate: " + icici.
getRateOfInterest());
        System.out.println("AXIS Interest Rate: " + axis.ge
tRateOfInterest());
    }
}
```

In this example:

- The `Bank` class defines a method `getRateOfInterest()`.
- Subclasses `SBI`, `ICICI`, and `AXIS` override the `getRateOfInterest()` method to provide their specific interest rates.
- When `getRateOfInterest()` is called on objects of different bank classes, the overridden method in each subclass is executed, providing the respective interest rate.




```
class A {
    void show() {
        System.out.println("I am in class A");
    }
}

class B extends A {
```

```
    void show() {
        System.out.println("I am in class B");
        // super.show(); // Uncomment this line to call the s
    }
}

class OverrideDemo {
    public static void main(String args[]) {
        A pobj = new A(); // Creating an object of class A
        B cobj = new B(); // Creating an object of class B
        A ref;

        ref = cobj; // Reference of type A referring to an ob
        ref.show(); // Calls the overridden show() method of
    }
}
```

```
I am in class B
```

Abstract class and Object class

Object class - Introduction:

- The `Object` class, defined in Java's `java.lang` package, serves as the root of the class hierarchy.
- It is a superclass of all other classes in Java, making it the most fundamental class.
- All classes implicitly inherit from the `Object` class if they do not explicitly specify another superclass.
- An object of type `Object` can refer to instances of any class, including arrays.

Object class Methods:

1. `getClass()`:
    - Returns the runtime class of the object.

- The returned `Class` object is the one locked by static synchronized methods of the represented class.

2. `hashCode()`:
   - Returns a hash code value for the object.
   - This method is used for hash tables like those in `HashMap`.
   - The hash code must consistently return the same integer when invoked on the same object during a single execution.
3. `equals(Object obj)`:
   - Indicates whether some other object is "equal to" this one.
   - It implements an equivalence relation on non-null object references:
     - Reflexive: `x.equals(x)` should return `true`.
     - Symmetric: `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
     - Transitive: If `x.equals(y)` and `y.equals(z)` both return `true`, then `x.equals(z)` should return `true`.
     - Consistent: Multiple invocations of `x.equals(y)` should consistently return `true` or consistently return `false`, as long as no information used in comparisons is modified.
     - For any non-null reference value `x`, `x.equals(null)` should return `false`.

```
public class ObjectExample {
    public static void main(String[] args) {
        Object obj1 = new Object();
        Object obj2 = new Object();

        System.out.println("getClass(): " + obj1.getCla
        System.out.println("hashCode(): " + obj1.hashCo
        System.out.println("equals(): " + obj1.equals(o
        System.out.println("toString(): " + obj1.toStri
```

```
    }
}
```

```
getClass(): class java.lang.Object
hashCode(): 1829164700
equals(): false
toString(): java.lang.Object@6d06d69c
```

```
class ObjectDemo {
    public static void main(String args[]) {
        Student s1 = new Student(10, "Ram");
        Student s2 = new Student(10, "Ram");

        System.out.println(s1.getClass()); // class Stu
        System.out.println(s1.hashCode());
        System.out.println(s1.equals(s2));
        System.out.println(s1 == s2);
    }
}

class Student {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    void display() {
        System.out.println("id=" + id);
        System.out.println("name=" + name);
    }
}
```

The difference between `equals()` method and `==` operator in Java is as follows:

1. `equals()` **method:**
    - `equals()` is a method defined in the `Object` class.
    - It is used to compare the contents of two objects to determine if they are equal.
    - By default, `equals()` method in the `Object` class checks for reference equality. However, it is overridden in many classes to compare the content of objects.
    - It returns `true` if the two objects are meaningfully equal, `false` otherwise.
    - Syntax: `public boolean equals(Object obj)`
2. `==` **operator:**
    - `==` operator is a comparison operator used to compare two objects' references.
    - It checks whether the two objects reference the same memory location or not.
    - It returns `true` if the references of two objects point to the same memory location, `false` otherwise.
    - It doesn't compare the content of objects; it only compares the memory addresses of the objects.

Here's how they are used:

```
class ObjectDemo {
    public static void main(String args[]) {
        Student s1 = new Student(10, "Ram");
        Student s2 = new Student(10, "Ram");

        System.out.println(s1.equals(s2)); // Check i
f objects are equal using equals() method
        System.out.println(s1 == s2);      // Check i
f references are equal using == operator
    }
}

class Student {
```

```
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Output:

```
false
false
```

Explanation:

- The `equals()` method returns `false` because it compares the content of objects, and even though the content is the same, `equals()` method is not overridden in the `Student` class.
- The `==` operator returns `false` because it compares the references of two objects, and `s1` and `s2` are two different objects with different memory addresses.

Abstract class:

- An abstract class is a class that cannot be instantiated directly; it serves as a blueprint for other classes to inherit from.
- It may contain abstract methods, which are declared but not implemented in the abstract class itself. Subclasses must provide implementations for these methods.
- Abstract classes can contain both abstract and concrete methods.
- Abstract classes are declared using the `abstract` keyword.

Example:

```
abstract class Shape {
    abstract void draw(); // Abstract method
```

```
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}

class Rectangle extends Shape {
    void draw() {
        System.out.println("Drawing a rectangle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape s1 = new Circle();
        Shape s2 = new Rectangle();
        s1.draw();
        s2.draw();
    }
}
```

In this example, `Shape` is an abstract class with an abstract method `draw()`. `Circle` and `Rectangle` are concrete subclasses of `Shape`, each providing its implementation of the `draw()` method.

toString():

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
    // Overriding toString method to provide a custom strin
g representation
    @Override
    public String toString() {
        return "Person[name=" + name + ", age=" + age +
"]";
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating a Person object
        Person person = new Person("John", 30);

        // Printing the object using toString method
        System.out.println(person); // Output: Person[name=
John, age=30]
    }
}
```

In this example:

Even though the `name` and `age` fields are `private`, the `toString()` method is defined within the `Person` class, which means it has access to all the `private` members of that class. This is because access control in Java is based on class, not object. Within the class definition itself, methods have access to all the private members of that class, regardless of whether they are directly accessing them or using getters/setters.

- The `toString()` method is overridden in the `Person` class to provide a custom string representation.
- When the `person` object is printed using `System.out.println()`, Java implicitly calls the `toString()` method on the object to get its string representation, which is then printed to the console.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Coding example – 1: Demo of overriding toString() function

```
class Box {
    int width;
    int height;
    int depth;
    Box()
    {   this.width = 0; this.height = 0; this.depth = 0;
    }
    Box(int l,int m,int n)
    {   this.width = l; this.height = m; this.depth = n;
    }
    @Override
    public String toString() {
        return width + " " + height + " " + depth;
    }
}
```

```
public class P1_object {
    public static void main(String[] args) {
        Box obj = new Box();
        System.out.println(obj);
        Box new_obj = new Box(3,2,1);
        System.out.println(new_obj);
    }
}
```

Coding example – 2: Demo of overriding of equals() function

```
class Box {
    int width;
    int height;
    int depth;
    Box()
    {   this.width = 0; this.height = 0; this.depth = 0;
    }
    Box(int l,int m,int n)
    {   this.width = l; this.height = m; this.depth = n;
    }
    @Override
    public boolean equals(Object o) {
        Box b2 = (Box) o; // imp
        return this.width == b2.width && this.height == b2.height && this.depth == b2.depth;
    }
}
```

```
public class P2_Object {
    public static void main(String[] args) {
        Box obj1 = new Box();
        Box obj2 = new Box(3,2,1);
        Box obj3 = new Box(3,2,1);
        System.out.println(obj1 == obj2);
        System.out.println(obj2.equals(obj3));
    }
}
```

In Java, the `==` operator checks if two objects reference the same memory location, i.e., if they are the same object. On the other hand, the `equals()` method is used to compare the contents of two objects to determine if they are meaningfully equivalent.

Here are some tricky cases involving `equals()` method in Java:

1. **String Literal vs String Object**: In Java, string literals are stored in the string pool, while string objects created using the `new` keyword are stored in the heap. So, using `==` to compare string literals may produce unexpected results, whereas `equals()` compares the content of the strings.

```
String str1 = "hello";
String str2 = new String("hello");

System.out.println(str1 == str2);          // Output: fa
lse
System.out.println(str1.equals(str2));    // Output: tru
e
```

2. **Overridden equals() Method**: For custom classes, the `equals()` method should be overridden to define meaningful equality. If `equals()` is not overridden, it behaves the same as `==`, comparing memory references.

```
class ObjectDemo {
    public static void main(String args[]) {
        Student s1 = new Student(10, "Ram");
        Student s2 = new Student(10, "Ram");

        System.out.println(s1.equals(s2)); // Check if o
bjects are equal using overridden equals() method
        System.out.println(s1 == s2);      // Check if r
eferences are equal using == operator
    }
}

class Student {
    int id;
    String name;
```

```
    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        return false;
    }
}
```

3. **Handling Null**: The `equals()` method should handle `null` gracefully to avoid NullPointerException.

```
String str = "hello";
System.out.println(str.equals(null));     // Output: fal
se
```

4. **Symmetry**: The `equals()` method should be symmetric, i.e., if `a.equals(b)` is true, then `b.equals(a)` should also be true.

```
String s1 = "hello";
String s2 = new String("hello");

System.out.println(s1.equals(s2));    // Output: true
System.out.println(s2.equals(s1));    // Output: true
```

Interface

In Java, an interface is a way to specify a set of method signatures that a class must implement. Here's an explanation and an example:

1. **Definition**: An interface in Java specifies the method signatures and has no default implementation. This means that the methods declared in an interface are abstract and public by default.

2. **Purpose**: Interfaces define the interaction between objects and the outside world. They specify a contract that classes must adhere to if they implement the interface. This allows for a level of abstraction and flexibility in the design of software systems.
3. **Example Interface**:

```
public interface Displayable {
    void disp();
}
```

In this example, `Displayable` is an interface with a single method `disp()`. Classes that implement this interface must provide an implementation for the `disp()` method.

1. **Implementing an Interface**:

```
class MyClass implements Displayable {
    @Override
    public void disp() {
        System.out.println("Displaying...");
    }
}
```

Here, `MyClass` implements the `Displayable` interface and provides an implementation for the `disp()` method specified in the interface.

1. **Benefits of Interfaces**:
    - **Abstraction**: Interfaces allow for abstraction by defining a contract without specifying the implementation details.
    - **Multiple Inheritance**: Java interfaces support multiple inheritance, allowing a class to implement multiple interfaces.
    - **Loose Coupling**: Interfaces promote loose coupling between components of a system by defining a common contract for interaction.

In this example, the `Bicycle` interface specifies the behavior that any bicycle should have. It declares four methods: `changeCadence`, `changeGear`, `speedUp`, and `applyBrakes`, which represent actions that can be performed on a bicycle.

To implement this interface, you would create a class representing a specific brand of bicycle, such as `AAABicycle`, and use the `implements` keyword to indicate that this class conforms to the `Bicycle` interface. By implementing the `Bicycle` interface, the `AAABicycle` class is required to provide implementations for all the methods declared in the interface.

Here's how the `AAABicycle` class would look like:

```
class AAABicycle implements Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    @Override
    public void changeCadence(int newValue) {
        cadence = newValue;
    }

    @Override
    public void changeGear(int newValue) {
        gear = newValue;
    }

    @Override
    public void speedUp(int increment) {
        speed = speed + increment;
    }

    @Override
    public void applyBrakes(int decrement) {
        speed = speed - decrement;
    }
}
```

In this class:

- The `implements Bicycle` clause indicates that `AAABicycle` implements the `Bicycle` interface.

- The class provides implementations for all the methods declared in the `Bicycle` interface.
- Each method overrides the corresponding method declared in the interface, providing specific behavior for the `AAABicycle` class.

By implementing the `Bicycle` interface, the `AAABicycle` class ensures that it adheres to the contract specified by the interface, allowing it to be treated as a bicycle and used interchangeably with other bicycle implementations.

Inheritance Vs Interface

PES

Category	Inheritance	Interface
Description	Inheritance is the mechanism in java by which one class is allowed to inherit the features of another class.	Interface is the blueprint of the class. It specifies what a class must do and not how. Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
Use	It is used to get the features of another class.	It is used to provide total abstraction.
Syntax	class subclass_name extends superclass_name { }	interface <interface_name>{ }
Number of Inheritance	It is used to provide 4 types of inheritance. (multi-level, simple and hierarchical inheritance)	It is used to provide 1 types of inheritance (multiple).
Keywords	It uses extends keyword.	It uses implements keyword.
Inheritance	We can inherit lesser classes than Interface if we use Inheritance.	We can inherit enormously more classes than Inheritance, if we use Interface.
Method Definition	Methods can be defined inside the class in case of Inheritance.	Methods cannot be defined inside the class in case of Interface (except by using **static** and **default** keywords).
Multiple Inheritance	We cannot do multiple inheritance (causes compile time error).	We can do multiple inheritance using interfaces.

1. **Instantiating an Interface**: No, you cannot instantiate an interface directly. Interfaces cannot have constructors, including default constructors.
2. **Data Members in an Interface**: Yes, interfaces can have data members, but they are implicitly static and final (immutable). These members are accessible through the class or object and exist in every class implementing the interface.
3. **Abstract Classes with All Methods Implemented**: Yes, a class with all methods implemented can still be abstract if creating an object of that class doesn't make sense in the application domain.
4. **Private Methods in an Interface**: No, methods in an interface are by default public and abstract. They cannot be private.
5. **Protected Methods in an Interface**: No, interfaces are meant to define contracts for all classes implementing them, so methods cannot be protected.

6. **Extending Interfaces**: Yes, an interface can extend another interface, allowing for hierarchical organization of interfaces.
7. **Implementing Multiple Interfaces**: Yes, a class can implement more than one interface. Since interfaces do not contain mutable members, there are no conflicts.
8. **Overriding Methods of an Interface**: If a class overrides a method of an interface, it remains abstract and cannot be instantiated directly.

Codes

```
interface Output {
    int y = 20;

    void disp();

    default void disphello() {
        System.out.println("hello");
    }
}

interface Display {
    void disp();
}

class Sample implements Output, Display {
    int x = 10;

    public void disp() {
        System.out.println("imple of disp");
        System.out.println(y);
    }

    public void disphello() {
        System.out.println("hello-overridden");
    }

    // instance method
```

```
    void print() {
        System.out.println("instance method");
    }
}

public class InterfaceDemo {
    public static void main(String args[]) {
        Sample s = new Sample();
        s.disp(); // interface method
        s.print(); // instance method
        System.out.println(s.y); // instance var
        System.out.println(Output.y);
        s.disp();
        System.out.println(Output.y); // to show y is public+
        s.disphello();
        Output obj = s; // to show interface ref can hold obj
        obj.disp();
    }
}
```

```
imple of disp
20
instance method
20
20
imple of disp
20
hello-overridden
imple of disp
20
```

```
interface Write {
    void disp();
    int c = 10;
    default void displayDefault() {
        System.out.println("I am a default method");
    }
```

```
}

interface Read {
    void disp();
}

class InterfaceDemo implements Write, Read {
    public void disp() {
        System.out.println("Implementation of disp");
    }

    public static void main(String args[]) {
        Write r = new InterfaceDemo();
        InterfaceDemo d = new InterfaceDemo();
        r.disp();
        System.out.println("Value of r= " + r.c);
        d.disp();
        System.out.println("Value of d= " + d.c);
        r.displayDefault();
    }
}
```

```
Implementation of disp
Value of r= 10
Implementation of disp
Value of d= 10
I am a default method
```

```
interface Polygon {
  void getArea(int length, int breadth);
}

// Implement the Polygon interface
class Rectangle implements Polygon {

  // Implementation of abstract method
  public void getArea(int length, int breadth) {
```

```
    System.out.println("The area of the rectangle is " + (len
  }
}

public class InterfaceDemo1 {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```

```
The area of the rectangle is 30
```

```
interface Shape{
    int d1 = 10, d2 = 20;   // final, static, and public
    void find_area(); // abstract, public
    default void hellomethod() {
        System.out.println("Hello from Shape!");
    }
}

class Rectangle implements Shape{
    int d1,d2;
    Rectangle(){}
    Rectangle(int d1,int d2)
    {
        this.d1 = d1;
        this.d2 = d2;
    }
    public void find_area() {
        System.out.println("Area of Rectangle: " + (d1 * d2))
    }
}

class Triangle implements Shape{
    int d1,d2,d3;
    Triangle(){}
```

```
    Triangle(int d1,int d2, int d3)
    {
        this.d1 = d1;
        this.d2 = d2;
        this.d3 = d3;
    }
    public void find_area() {
        System.out.println("Area of Triangle");
    }
}

class Circle implements Shape{
    public void find_area() {
        System.out.println("Area of Circle");
    }
}

public class InterfaceDemo2 {
    public static void main(String[] args) {
        Shape s1 = new Rectangle();
        s1.find_area(); // Output: Area of Rectangle: 200
        s1.hellomethod(); // Output: Hello from Shape!
        System.out.println(s1.d1); // Output: 10
        // s1.d1 = 100; // Compile-time error, d1 is final

        Rectangle r = new Rectangle(5,6);
        // r.d1 = 100; // Valid, d1 is not accessed through s
        System.out.println(s1.d1+" "+r.d1); // Output: 10 5

        Rectangle r2 = new Rectangle(5,6);
        s1 = r2;
        // r.d1 = 500; // Valid, d1 is not accessed through s
        System.out.println(s1.d1); // Output: 10
    }
}
```

```
Area of Rectangle: 200
Hello from Shape!
```

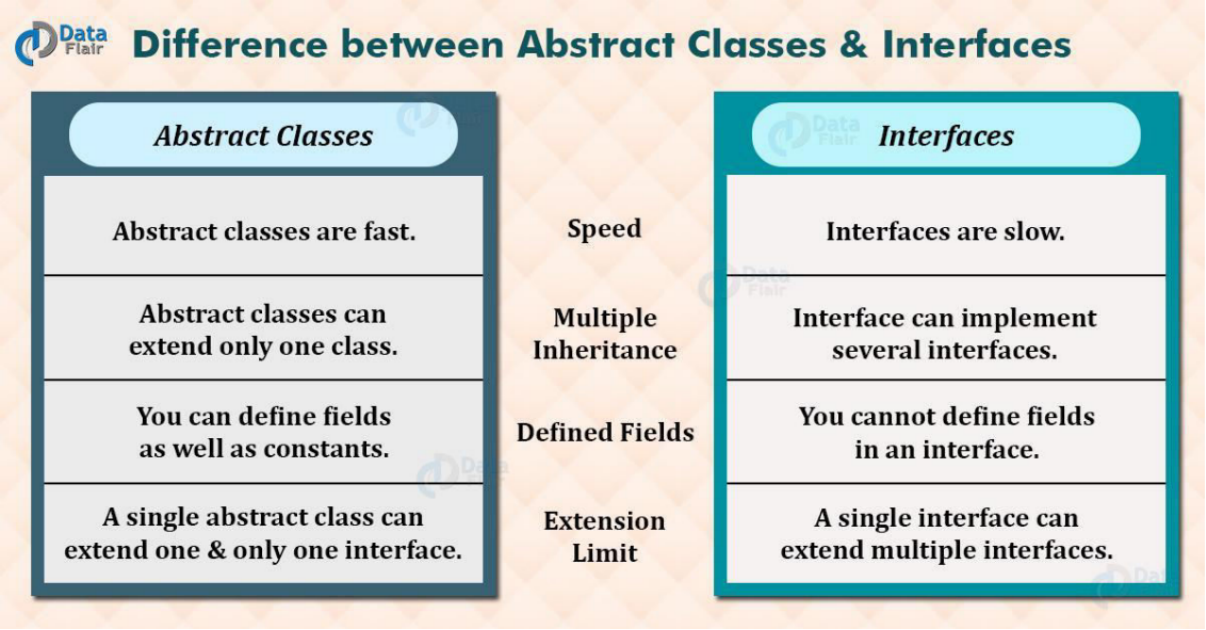
```
10
10 5
10
```

1. `Shape` is an interface, and `Rectangle` is a class that implements the `Shape` interface. According to polymorphism, you can use an interface type to refer to a subclass object.
2. The line `Shape s1 = new Rectangle();` creates an object of the `Rectangle` class and assigns it to a reference variable `s1` of the `Shape` type. This is possible because a `Rectangle` is a `Shape` (due to inheritance), so you can use a `Shape` reference to refer to a `Rectangle` object.
3. After this assignment, you can only call methods and access variables that are defined in the `Shape` interface, even though the object being referred to is actually a `Rectangle`. This is known as polymorphism.

Abstract classes

1. **Purpose of Abstract Classes**:
    - Abstract classes provide implementation reuse by defining a generalized form shared by all subclasses.
    - They determine the nature of methods that subclasses must implement, known as subclasser responsibility.
    - Abstract classes have no method body.
2. **Creation and Usage**:
    - Abstract classes are created using the `abstract` keyword at the beginning of the class declaration.
    - They may contain abstract methods, which are methods without a body.
    - If a class has at least one abstract method, it must be declared abstract.
    - Abstract classes cannot be instantiated directly; they serve as templates for other classes.
    - When a class extends an abstract class, it must either provide implementations for all abstract methods or declare itself as abstract.

- Abstract classes can have both static and non-static data members and methods like any other Java class.
- Abstract classes cannot be declared as `final` because they are meant to be extended.
- Java does not support multiple inheritance, so a class can extend only one abstract class.




```
// Abstract class
abstract class Shape {
    // Abstract method
    abstract double area();

    // Concrete method
    void display() {
        System.out.println("This is a shape.");
    }
}

// Concrete subclass
class Rectangle extends Shape {
    double length;
    double width;
```

```
    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    // Implementation of abstract method
    double area() {
        return length * width;
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(5, 4);
        System.out.println("Area of rectangle: " + rectangl
e.area());
        rectangle.display();
    }
}
```

In this example, `Shape` is an abstract class with an abstract method `area()` and a concrete method `display()`. The `Rectangle` class extends `Shape` and provides an implementation for the `area()` method.

```
abstract class Shape {
    abstract void find_area();

    void helloMethod() {
        System.out.println("in helloMethod");
    }
}

class Rectangle extends Shape {
    int d1, d2;

    Rectangle() {
    }
```

```
    Rectangle(int d1, int d2) {
        this.d1 = d1;
        this.d2 = d2;
    }

    void find_area() {
        System.out.println("Area of Rectangle: " + (d1 * d2))
    }
}

class Triangle extends Shape {
    int d1, d2, d3;

    Triangle() {
    }

    Triangle(int d1, int d2, int d3) {
        this.d1 = d1;
        this.d2 = d2;
        this.d3 = d3;
    }

    void find_area() {
        System.out.println("Area of Triangle");
    }
}

class Circle extends Shape {
    void find_area() {
        System.out.println("Area of Circle");
    }
}

public class AbstractDemo {
    public static void main(String[] args) {
        Shape s1 = new Rectangle(10, 20);
        s1.find_area(); // Output: Area of Rectangle: 200
```

```
        s1.helloMethod(); // Output: in helloMethod

        Shape s2 = new Triangle(100, 200, 300);
        s2.find_area(); // Output: Area of Triangle
        s2.helloMethod(); // Output: in helloMethod

        Shape s3 = new Circle();
        s3.find_area(); // Output: Area of Circle
        s3.helloMethod(); // Output: in helloMethod
    }
}
```

```
class MainAbstract {
    public static void main(String[] args) {
        Dog myDog = new Dog(); // Create a Dog object
        myDog.animalSound();
        myDog.sleep();
    }
}

abstract class Animal {        // Abstract class
    public abstract void animalSound();      // Abstract meth

    public void sleep() {            // Regular method
        System.out.println("Zzz");
    }
}

class Dog extends Animal {        // Subclass (inherit from A
    public void animalSound() {
        System.out.println("The Dog says: bow bow ");    // T
    }
}
```

Now, let's compare abstract classes with interfaces:

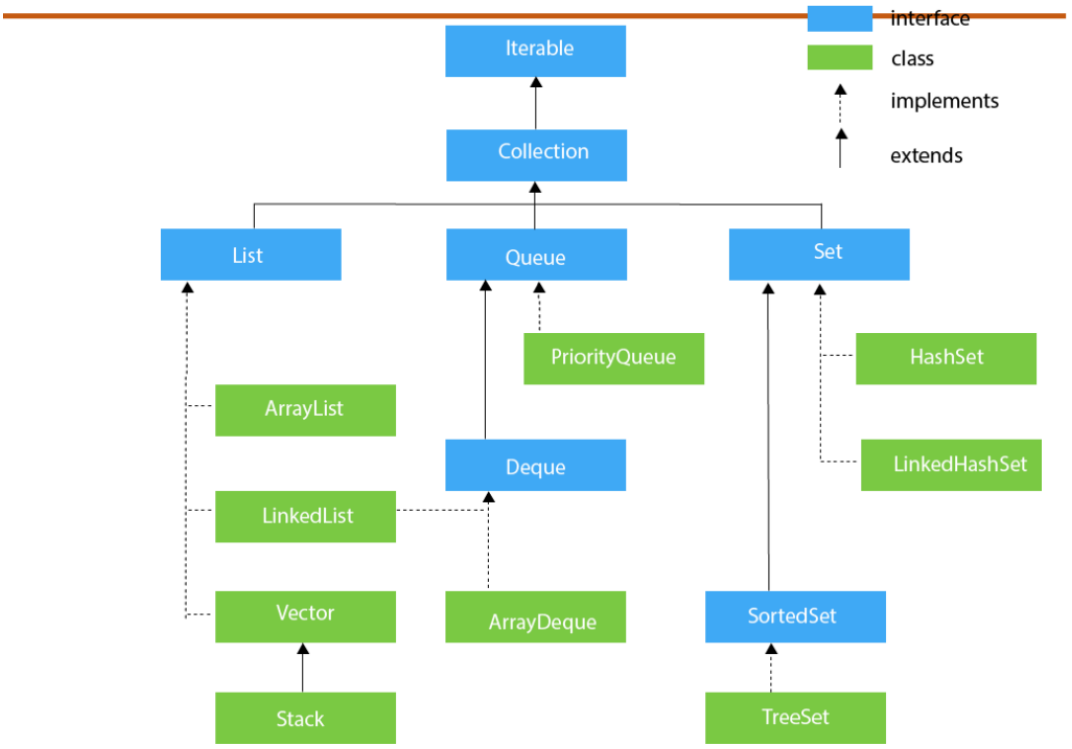
1. **Abstract Classes**:

- Abstract classes can have abstract methods (methods without implementation) as well as concrete methods.
- Abstract classes can have constructors.
- They can have both static and non-static methods and data members.
- A class can extend only one abstract class.

2. **Interfaces**:
   - Interfaces can only declare abstract methods; they cannot contain any method implementations.
   - Interfaces cannot have constructors.
   - All methods in an interface are implicitly public and abstract.
   - Interfaces can only have static final variables, not any other kind of variable.
   - A class can implement multiple interfaces.

Collections

1. The Java Collections Framework offers ready-made data structures and algorithms to represent groups of objects.
2. It simplifies development by providing high-performance implementations of these data structures and algorithms.
3. Interoperability is enhanced through a common language for passing collections between different parts of a program or between different programs.
4. Developers benefit from reduced learning curves as they only need to learn standardized collection interfaces and methods.
5. The framework fosters software reuse by promoting the use of standardized interfaces and algorithms across projects.



Array - Introduction:

- Arrays are collections of similar elements stored in contiguous memory allocation.
- They can be of any type and come in one-dimensional or two-dimensional forms.
- Declaration and instantiation follow specific syntax, allowing for the creation of arrays of desired sizes and types.

Instantiation:

- Arrays are instantiated using the syntax: `arrayRefVar = new datatype[size];`
- For example, `int[][] arr = new int[3][3];` creates a 2D array with 3 rows and 3 columns.

- Declaration of 1D Array & Instantiation:
  - dataType[ ] arr; (or)
  - dataType [ ]arr; (or)
  - dataType arr[ ];

Instantiation:

arrayRefVar=**new** datatype[size];

- Declaration of 2D Array & Instantiation: :
  - dataType[ ][ ] arr; (or)
  - dataType [ ][ ]arr; (or)
  - dataType arr [ ][ ];

Instantiation:

int[][] arr=**new int**[3][3]; //3 rows and 3 columns

Array Representation:

- Arrays can be represented visually, showing the layout of elements in memory for both one-dimensional and two-dimensional arrays.

Foreach Loop:

- The foreach loop iterates over array elements one by one, holding each element in a variable for processing.
- Its syntax is `for(data_type variable: array){ ... }`.
- Arrays class in java.util package is a part of the Java Collection Framework.
- It provides static methods to dynamically create and access Java arrays.
- It consists of only static methods and the methods of Object class. The methods of this class can be used by the class name itself.
  ● The class hierarchy is as follows:
  - java.lang.Object
  - java.util.Arrays

Arrays Class in Java:

- The `Arrays` class in `java.util` package provides static methods for dynamically creating and accessing Java arrays.
- It consists only of static methods and inherits methods from the `Object` class.
- Commonly used methods include `binarySearch()`, `asList()`, `toString()`, `equals()`, and `sort()`.

```
class ForEach {
    public static void main(String args[]) {
        int arr[] = {12, 13, 14, 44};
        int total = 0;
        for(int i : arr) {
            total = total + i;
        }
        System.out.println("Total: " + total);  //Total: 83
    }
}
```

Introduction to List Interface in Java:

- The List interface in Java provides a means to maintain ordered collections.
- It supports index-based methods for inserting, updating, deleting, and searching elements, allowing for duplicate elements and storing null values.
- Found in the `java.util` package, the List interface inherits from the Collection interface.
- Implementation classes of List include ArrayList, LinkedList, Stack, and Vector.

Declarations:

- The List interface is declared as `public interface List<E> extends Collection<E>`.

Few Methods:

- Common methods provided by the List interface include `size()`, `clear()`, `add()`, `addAll()`, `contains()`, `containsAll()`, `equals()`, `hashCode()`, `isEmpty()`, `indexOf()`, and more.

```
import java.util.ArrayList;
import java.util.List;

public class ListExample {
public static void main(String[] args) {
// Creating a List of Strings
```

```
List<String> myList = new ArrayList<>();
// Adding elements to the list
    myList.add("Apple");
    myList.add("Banana");
    myList.add("Orange");

    // Displaying the list size
    System.out.println("Size of the list: " + myList.size
());

    // Displaying all elements in the list
    System.out.println("Elements in the list:");
    for (String fruit : myList) {
        System.out.println(fruit);
    }

    // Checking if the list contains a specific element
    String searchItem = "Banana";
    if (myList.contains(searchItem)) {
        System.out.println(searchItem + " is found in the l
ist.");
    } else {
        System.out.println(searchItem + " is not found in t
he list.");
    }
}
}
```

```
import java.util.*;

public class ArrayList1 {
    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<String>(); //
        list.add("Mango"); // Adding objects to ArrayList
        list.add("Apple");
        list.add("Banana");
        list.add("Grapes");
```

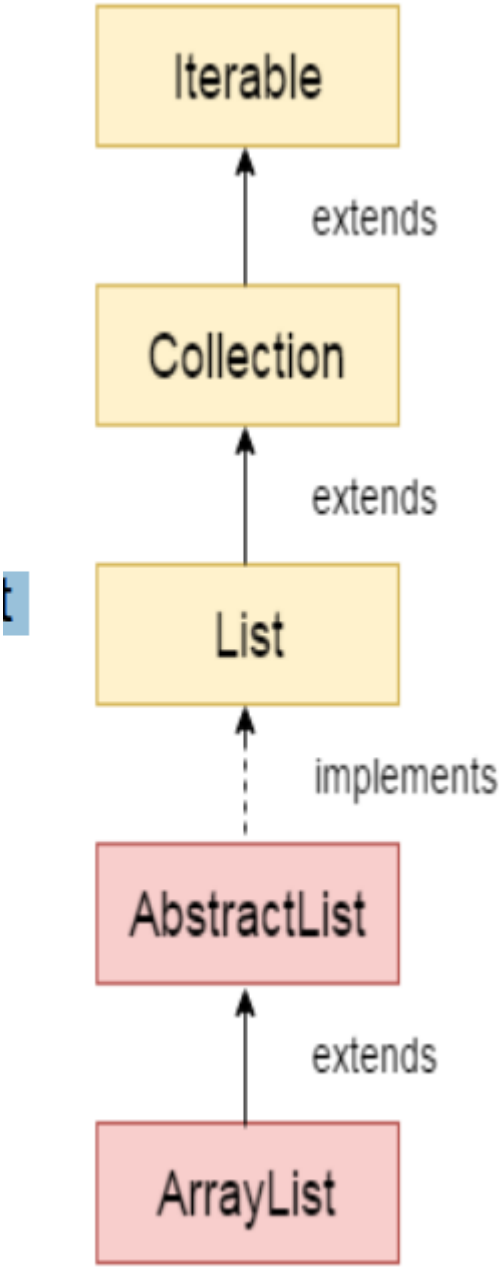
```
        // Traversing list through for-each loop
        for(String fruit : list) {
            System.out.println(fruit);
        }
    }
}
```

Introduction to ArrayList Class in Java:

- The ArrayList class utilizes a dynamic array for storing elements.
- It imposes no size limit and allows for the addition or removal of elements at any time.
- Located in the `java.util` package, it permits duplicate elements.
- ArrayList implements the List interface, enabling the use of all List interface methods.
- Maintains the insertion order of elements.
- Not suitable for creating ArrayLists of primitive types like int, float, or char; instead, wrapper classes must be used.

Example:

```
// Does not work
ArrayList<int> al = ArrayList<int>();

// Works fine
ArrayList<Integer> al = new ArrayList<Integer>();
```




```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Creating an ArrayList of strings
        ArrayList<String> list = new ArrayList<String>();

        // Adding elements to the ArrayList
        list.add("Apple");
```

```
        list.add("Banana");
        list.add("Orange");

        // Displaying the ArrayList
        System.out.println("ArrayList: " + list);

        // Adding an element at a specific index
        list.add(1, "Mango");

        // Displaying the modified ArrayList
        System.out.println("Modified ArrayList: " + list);

        // Removing an element
        list.remove("Banana");

        // Displaying the ArrayList after removal
        System.out.println("ArrayList after removal: " + li
st);
    }
}
```

Output:

```
ArrayList: [Apple, Banana, Orange]
Modified ArrayList: [Apple, Mango, Banana, Orange]
ArrayList after removal: [Apple, Mango, Orange]
```

This example demonstrates the creation, modification, and removal of elements in an ArrayList.

```
import java.util.*;
import java.io.*;

@SuppressWarnings("unused")
class ArrayList2 {
    public static void main(String[] args) {
        ArrayList<String> animals = new ArrayList<>();
```

```
        // add elements to the array list
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("ArrayList: " + animals);

        // get the element at index 1
        String str = animals.get(1);
        System.out.println("Element at index 1: " + str);

        // change element at index 2
        animals.set(2, "Goat");
        System.out.println("Updated ArrayList: " + animals);

        // remove element from index 2
        str = animals.remove(2);
        System.out.println("Updated ArrayList after removing:
        System.out.println("Removed Element: " + str);

        // sort the ArrayList
        Collections.sort(animals);
        System.out.println("After sorting animals:");
        System.out.println(animals);

        // ArrayList
        ArrayList<String> food = new ArrayList<String>();
        food.add("Biriyani");
        food.add("pizza");
        food.add("banana");
        food.add("curry");

        // iterate through the ArrayList using for loop
        for(int i = 0; i < food.size(); i++) {
            System.out.println(food.get(i));
        }
        System.out.println("\n");

        // update and remove elements from ArrayList
```

```
        food.set(0, "Chicken Biriyani");
        food.remove(2);
        // food.clear(); // uncomment to clear the ArrayList
        for(int j = 0; j < food.size(); j++) {
            System.out.println(food.get(j));
        }
    }
}
```

```
ArrayList: [Dog, Cat, Horse]
Element at index 1: Cat
Updated ArrayList: [Dog, Cat, Goat]
Updated ArrayList after removing: [Dog, Cat]
Removed Element: Goat
After sorting animals:
[Cat, Dog]
Biriyani
pizza
banana
curry


Chicken Biriyani
pizza
curry
```

Feature	Array	ArrayList
Declaration	Fixed size	Dynamic size
Initialization	Requires predefined size	No predefined size required
Storage	Contiguous memory allocation	Non-contiguous memory allocation
Memory Efficiency	Efficient in terms of memory	Less memory efficient due to internal storage
Access Methods	Direct index access	Methods like get() and set() for access
Length Retrieval	Uses length variable	Uses size() method

Automatic Resizing	Not applicable	Automatically resized when needed
Performance	Better performance for small, fixed-size collections	Slower performance due to dynamic resizing
Type Safety	Limited by primitive type or Object type only	Fully supports generics, ensuring type safety
Compatibility	Supports primitives and objects	Supports only objects (autoboxing for primitives)
Syntax	Primitive type[] arrayName; Object[] arrayName;	ArrayList<Type> arrayListName;

Stack

The `Stack` class in Java is a part of the Java Collection Framework and is used to implement the Stack data structure. A stack follows the Last-In-First-Out (LIFO) principle, meaning that the element that is added last is the first one to be removed.

Declaration:

```
public class Stack<E> extends Vector<E>
```

- `E` is the type of elements in the Stack.

Implemented interfaces:

- `Serializable`: A marker interface that classes must implement if they are to be serialized and deserialized.
- `Cloneable`: An interface that needs to be implemented by a class to allow its objects to be cloned.
- `Iterable<E>`: An interface representing a collection of objects that can be iterated.
- `Collection<E>`: A group of objects known as its elements. It represents a collection of objects.
- `List<E>`: Provides a way to store an ordered collection. It's a child interface of `Collection`.

- `RandomAccess`: A marker interface used by `List` implementations to indicate that they support fast random access.

Creation of a `Stack` class:

To create a `Stack` object, you need to import the `java.util.Stack` package and use the `Stack()` constructor.

Few Methods:

- `empty()`: Checks if the stack is empty.
- `push(E item)`: Pushes an item onto the top of the stack.
- `pop()`: Removes the object at the top of the stack and returns that object.
- `peek()`: Looks at the object at the top of the stack without removing it from the stack.
- `search(Object o)`: Searches for an object in the stack and returns its position from the top of the stack.

Example:

Here is an example demonstrating the use of `Stack` class:

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        // Creating a stack
        Stack<Integer> stack = new Stack<>();

        // Pushing elements onto the stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // Popping elements from the stack
        int poppedElement = stack.pop();
        System.out.println("Popped Element: " + poppedEleme
nt);
```

```
        // Peeking at the top of the stack
        int topElement = stack.peek();
        System.out.println("Top Element: " + topElement);

        // Checking if the stack is empty
        boolean isEmpty = stack.empty();
        System.out.println("Is stack empty? " + isEmpty);
    }
}
```

Output:

```
Popped Element: 30
Top Element: 20
Is stack empty? false
```

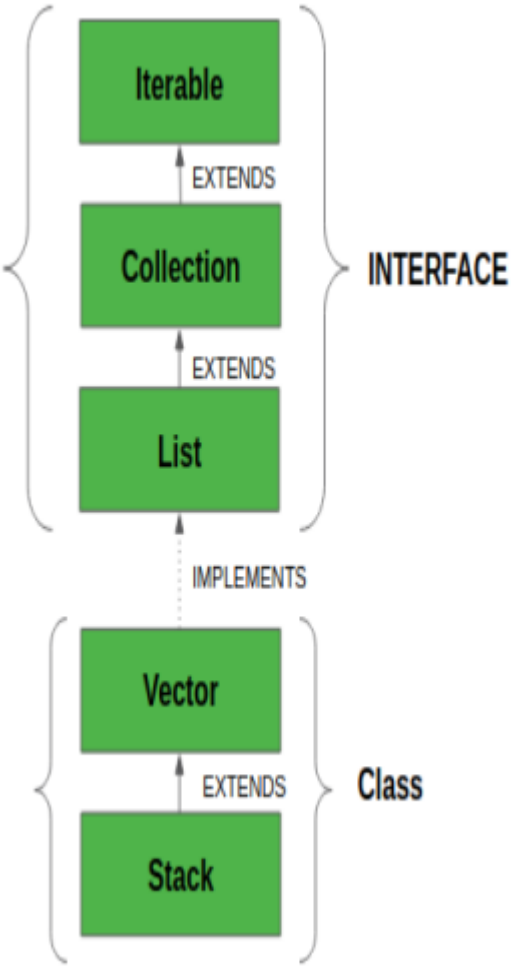
```
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        // Creating a Stack
        Stack<String> stack = new Stack<>();

        // Pushing elements onto the stack
        stack.push("Apple");
        stack.push("Banana");
        stack.push("Orange");

        // Displaying the stack
        System.out.println("Stack: " + stack);

        // Popping an element from the stack
        String poppedElement = stack.pop();
        System.out.println("Popped Element: " + poppedEleme
nt);

        // Displaying the stack after popping
```

```
        System.out.println("Stack after popping: " + stac
k);

        // Peeking the top element of the stack
        String topElement = stack.peek();
        System.out.println("Top Element: " + topElement);

        // Searching for an element in the stack
        int index = stack.search("Banana");
        System.out.println("Index of 'Banana': " + index);
    }
}
```

Output:

```
Stack: [Apple, Banana, Orange]
Popped Element: Orange
Stack after popping: [Apple, Banana]
Top Element: Banana
Index of 'Banana': 1
```

This example demonstrates the creation, pushing, popping, peeking, and searching operations on a Stack in Java.



Architectural Patterns

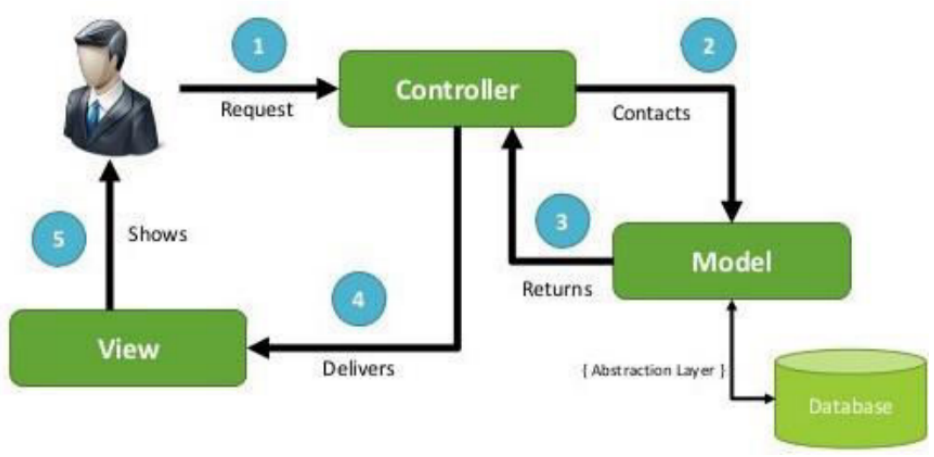
Architectural Pattern:

- A general, reusable solution to commonly occurring problems in software architecture.
- Helps define the basic characteristics and behavior of an application.
- Some patterns are highly scalable, while others are more agile, depending on specific business needs.
- Addresses issues like hardware performance limitations, high availability, and risk minimization.

MVC (Model-View-Controller):

- Introduced by Trygve Reenskaug in 1979.

- Decouples data access and business logic from presentation.
- Divides application into three main logical components: model, view, and controller.
- Separates application logic from presentation.
- Distinguishes between data model, processing control, and user interface.
- Neatly separates graphical interface from code managing user actions.
- Completely separates calculations and interface from each other.



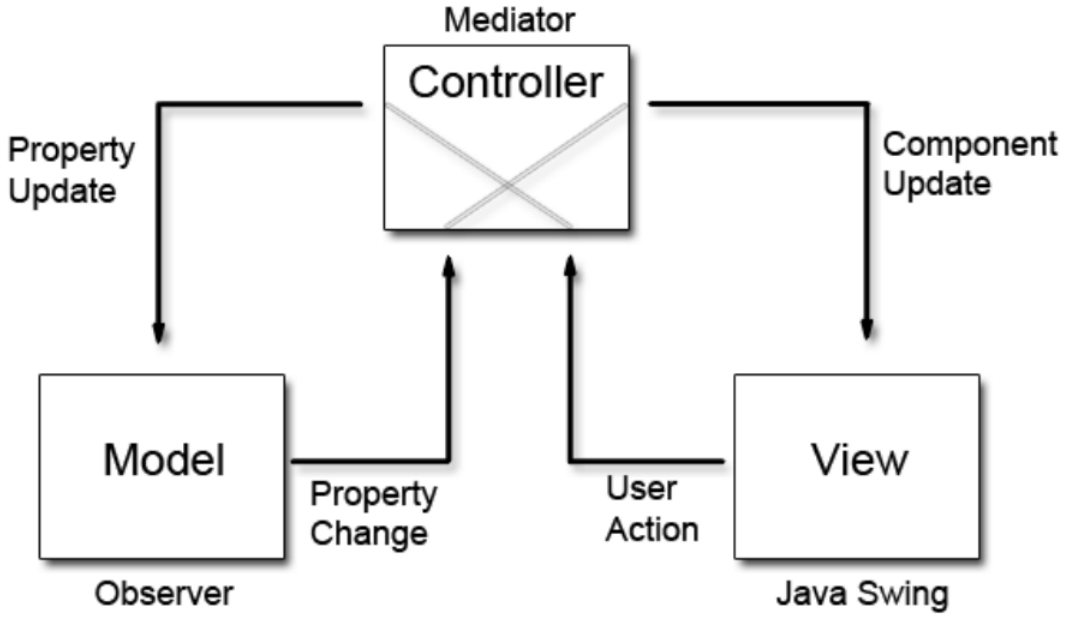
The MVC (Model-View-Controller) pattern is a well-known design pattern in web development that provides a structured way to organize code. It consists of three main components:

1. **Model:**
    - Represents data and the rules governing access to and updates of this data.
    - Often serves as a software approximation of a real-world process in enterprise software.
    - Defines the structure and behavior of data.
    - Manages the application's data, logic, and rules.
2. **View:**
    - Renders the contents of the model.
    - Specifies how the model data should be presented to the user.
    - Represents the presentation layer of the application.

- Visualizes the data contained in the model.
- Updates its presentation in response to changes in the model's data.

3. **Controller:**
    - Translates user interactions with the view into actions to be performed by the model.
    - Handles user input and performs appropriate actions based on that input.
    - Initiates changes in the model's state based on user actions.
    - May select a new view to present back to the user based on the context.
    - Acts as an intermediary between the model and the view.

In summary, the MVC pattern separates an application into three distinct components – model, view, and controller – each with its own responsibilities. This separation of concerns promotes modularity, maintainability, and reusability of code. The model represents data and business logic, the view represents the user interface, and the controller manages user input and updates the model accordingly. This architectural pattern helps developers build scalable, organized, and maintainable web applications.



Advantages of MVC

- Faster Development Process
- Model can be reused with multiple views
- Less dependency among components – loose coupling
- The modification does not affect the entire model
- Application becomes more understandable
- Not a single massive codebase

MVC in Java

The Model-View-Controller (MVC) architecture in Java consists of three main components:

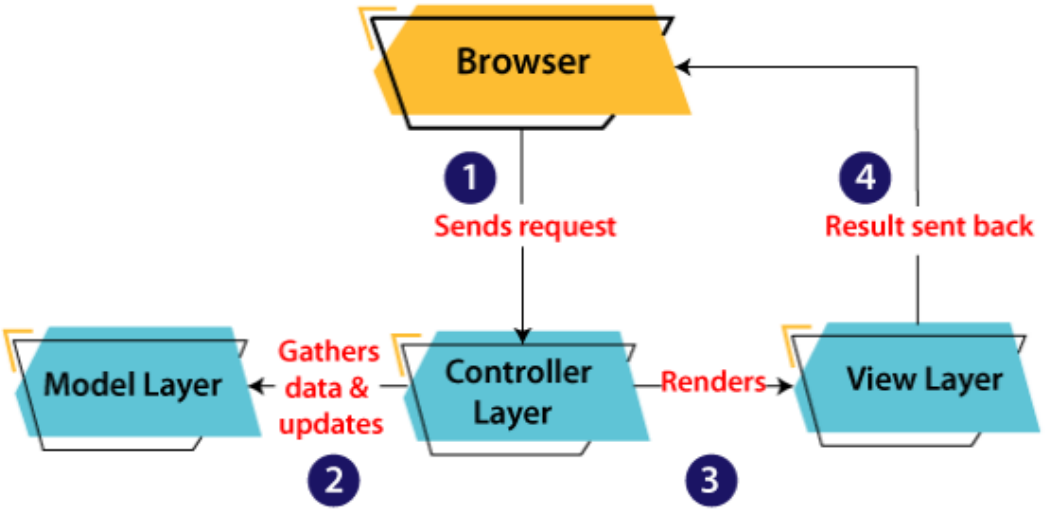
1. **Model:**
    - Contains simple Java classes that represent the application's data and business logic.
    - Manages the application's data, rules, and behavior.
    - Performs operations such as data retrieval, manipulation, and storage.
2. **View:**
    - Responsible for displaying the data to the user.
    - Presents the information obtained from the model in a user-friendly format.
    - Renders the user interface elements such as web pages, forms, or GUI components.
3. **Controller:**
    - Contains servlets or other components responsible for handling user requests and interactions.
    - Acts as an intermediary between the model and the view.
    - Processes user input, initiates changes in the model, and updates the view accordingly.

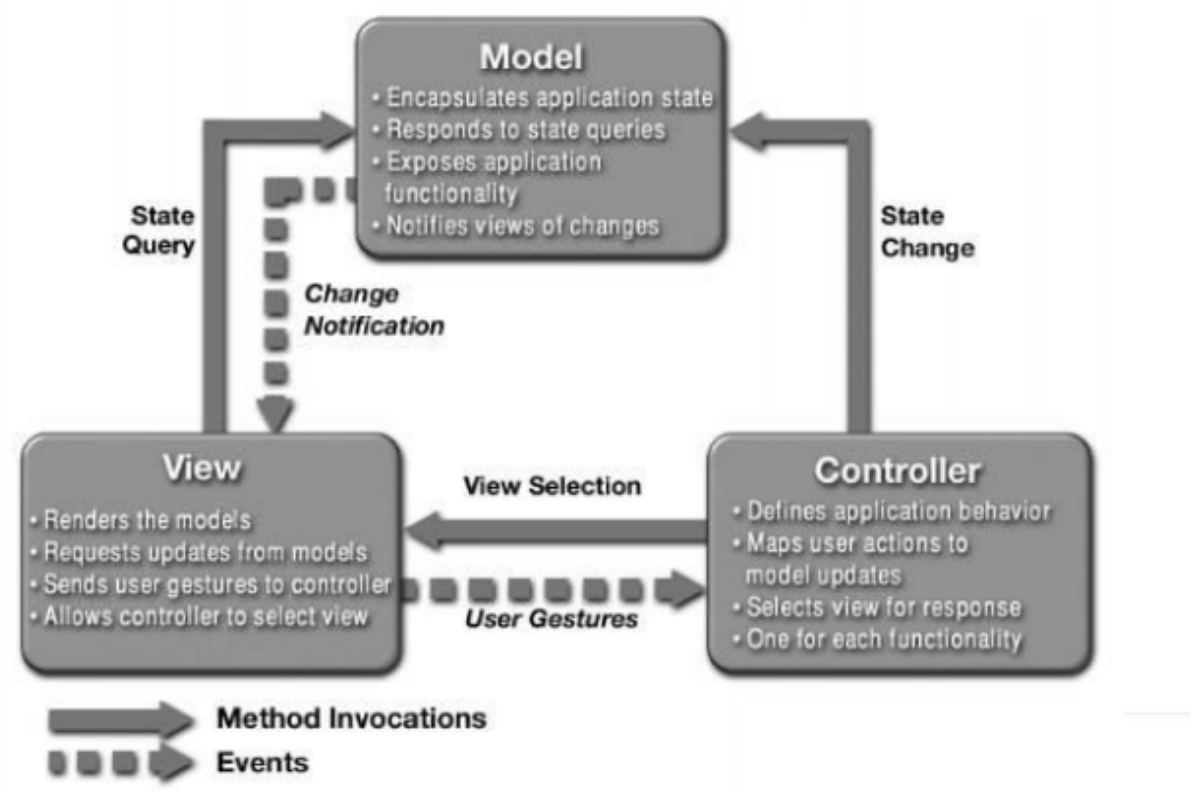
The MVC architecture separates the concerns of an application, promoting modularity, maintainability, and reusability of code. The flow of data and control in an MVC application typically follows these steps:

MVC working in Java

1. **Client Request:**
    - A client, such as a web browser, sends a request to the server for a specific page or resource.
2. **Controller Processing:**
    - The controller on the server side receives the request and determines the appropriate action to take.
    - It interacts with the model to gather the required data and perform necessary operations.
3. **Data Transfer to View:**
    - Once the controller has processed the request and obtained the necessary data from the model, it transfers this data to the view.
4. **View Rendering:**
    - The view receives the data from the controller and renders it to produce the desired output.
    - This output is typically in the form of HTML pages, GUI screens, or other user interface elements.
5. **Response to Client:**
    - The resulting output generated by the view is sent back to the client (browser) for display to the user.

One of the key benefits of the MVC architecture is that multiple views can interact with the same underlying model. This allows for flexibility and reusability in designing different user interfaces for the same application logic.





Example of MVC

In the MVC architecture described for the Employee management system:

1. **Employee Class (Model Layer):**
    - Represents the business logic and state of the application.
    - Handles fetching and storing of data in the database.

- Applies rules to the data, enforcing the application's concepts and logic.

2. **EmployeeView Class (View Layer):**
   - Represents the visualization of data received from the model.
   - Generates the output or user interface of the application.
   - Sends requested data to the client, retrieved from the model layer by the controller.
3. **EmployeeController Class (Controller Layer):**
   - Acts as an intermediary between the model and view layers.
   - Receives user requests from the view layer and processes them, including necessary validations.
   - Initiates data processing by interacting with the model layer based on user requests.
   - Receives processed data from the model layer and then displays it on the view.

Serialization

Serialization is the process of converting an object into a sequence of bytes, which can be persisted to a file or sent over the network. Deserialization is the reverse process where the serialized bytes are converted back into an object.

Here's a Java program that demonstrates serialization and deserialization of a `Student` object:

```
import java.io.*;

class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int rollNumber;

    public Student(String name, int rollNumber) {
        this.name = name;
```

```
        this.rollNumber = rollNumber;
    }

    public String getName() {
        return name;
    }

    public int getRollNumber() {
        return rollNumber;
    }
}

public class SerializationDemo {
    public static void main(String[] args) {
        // Creating a Student object
        Student student = new Student("John", 101);

        // Serialization
        try {
            // Saving the object to a file
            FileOutputStream fileOut = new FileOutputStream
("student.ser");
            ObjectOutputStream out = new ObjectOutputStream
(fileOut);
            out.writeObject(student);
            out.close();
            fileOut.close();
            System.out.println("Student object serialized s
uccessfully.");
        } catch (IOException i) {
            i.printStackTrace();
        }

        // Deserialization
        Student deserializedStudent = null;
        try {
            // Reading the object from a file
            FileInputStream fileIn = new FileInputStream("s
```

```
tudent.ser");
            ObjectInputStream in = new ObjectInputStream(fi
leIn);
            deserializedStudent = (Student) in.readObject
();
            in.close();
            fileIn.close();
            System.out.println("Student object deserialized
successfully.");
        } catch (IOException | ClassNotFoundException i) {
            i.printStackTrace();
        }

        // Displaying deserialized student details
        if (deserializedStudent != null) {
            System.out.println("Deserialized Student Detail
s:");
            System.out.println("Name: " + deserializedStude
nt.getName());
            System.out.println("Roll Number: " + deserializ
edStudent.getRollNumber());
        }
    }
}
```

This program does the following:

1. Creates a `Student` object.
2. Serializes the `Student` object to a file named `student.ser`.
3. Deserializes the `Student` object from the file.
4. Prints the details of the deserialized `Student` object.

Key methods used in serialization and deserialization:

- For serialization:
  - `ObjectOutputStream.writeObject(Object obj)`: Writes the specified object to the ObjectOutputStream.
  - `FileOutputStream`: Used to write data to a file.

- For deserialization:
  - `ObjectInputStream.readObject()`: Reads the next object from the ObjectInputStream.
  - `FileInputStream`: Used to read data from a file.