# **Spring AOP:**

## Why Use Spring AOP?

* **Separation of Concerns:** It allows you to separate business logic from cross-cutting concerns.
* **Reusability:** You can apply the same aspect across multiple methods or classes without duplicating code.
* **Maintainability:** It makes your code more modular and easier to maintain.

**Summary**

* **Aspect** encapsulates the cross-cutting concern.
* **Advice** defines what action to take and when.
* **Join Point** is where the action is applied.
* **Pointcut** specifies the join points to which the advice applies.
* **Weaving** applies aspects to target objects, enabling the separation of concerns.

# **Java:**

## **Java is not a fully object-oriented language**

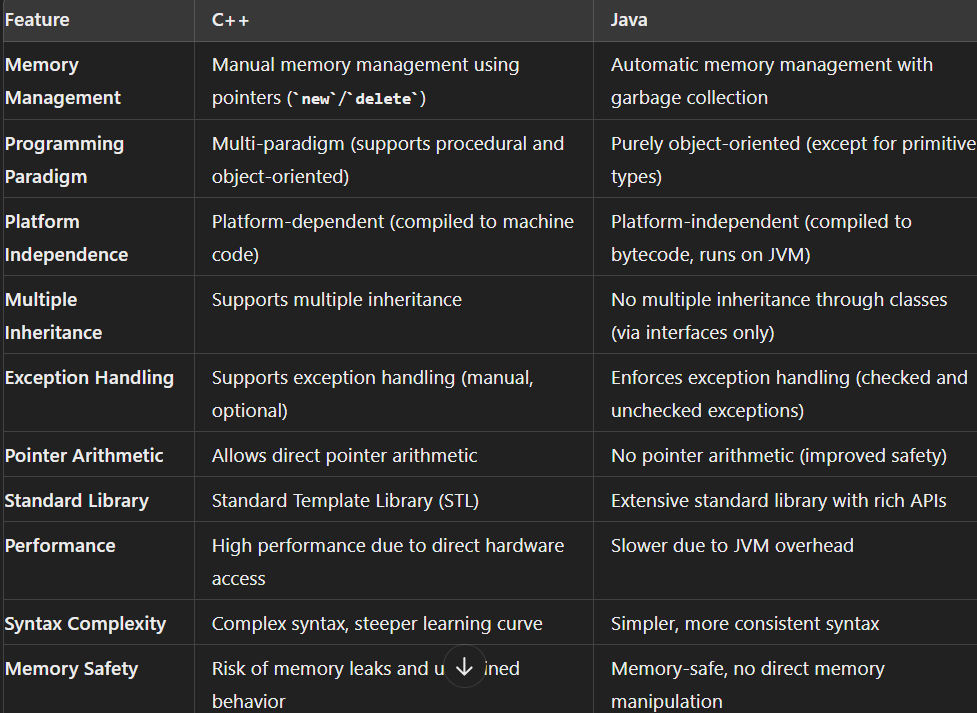
as it supports primitive data types like int, byte, long, short, etc., which are not objects. Hence these data types like int, float, double, etc., are not object-oriented. That's why Java is not 100% object-oriented

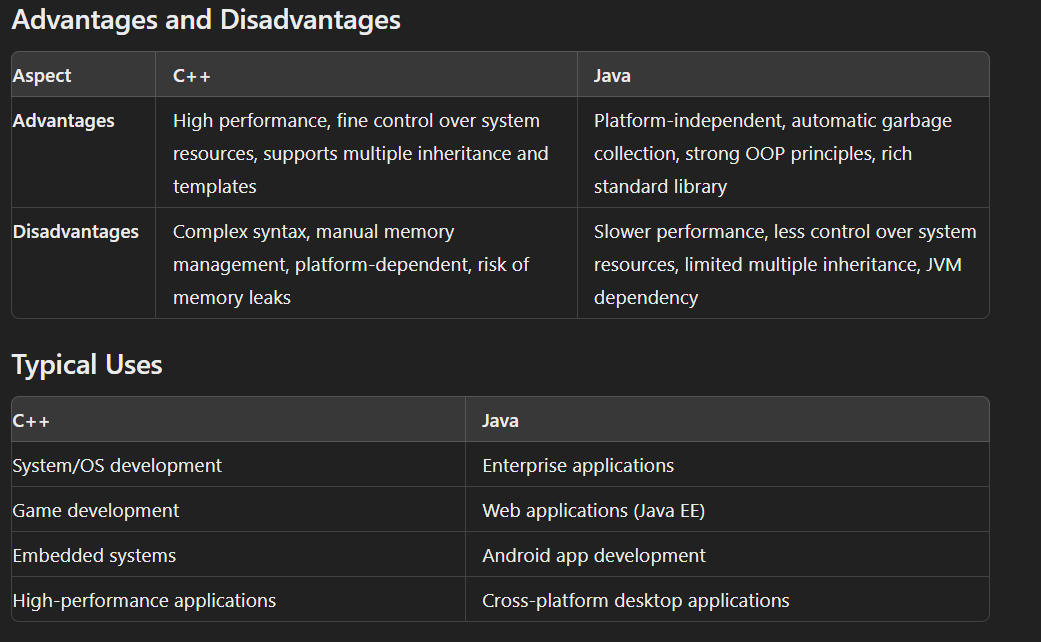
## **Heap memory**

is the portion that was not allocated to the java program but it will be available for use by the java program when it is required, mostly during the runtime of the program.

* When we write a java program then all the variables, methods, etc are stored in the **stack memory.**
* And when we create any object in the java program then that object was created in the **heap memory**. And it was referenced from the stack memory.

# **C++ vs Java**





In Java, the **Diamond Problem** is a common issue that arises with multiple inheritance. When a class inherits from multiple parent classes that have the same method or field, the compiler gets confused on which one to choose. This can lead to unexpected behavior and errors in the program.

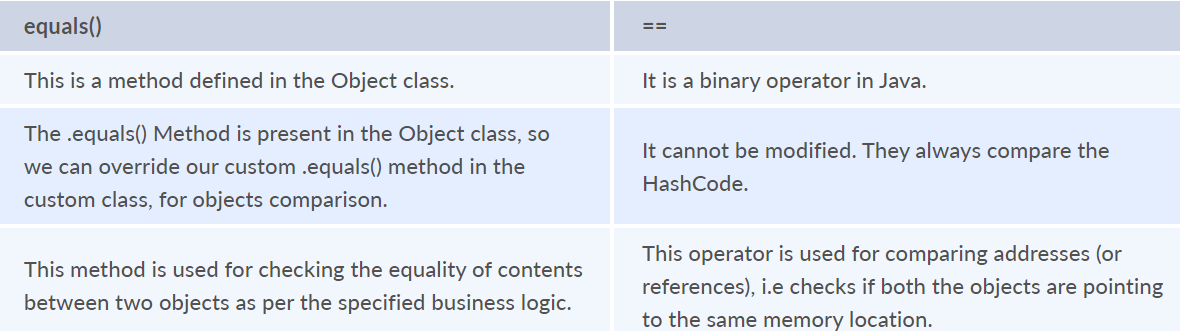
**where there are objects present in the heap that are no longer used, but the garbage collector is unable to remove them from memory,** and therefore, they’re unnecessarily maintained

# **JIT compiler**

The compiler is nothing but a translator of source code to machine-executable code

Working:

* First, the Java source code (.java) conversion to byte code (.class) occurs with the help of the Javac compiler.
* Then, the .class files are loaded at run time by JVM, and with the help of an interpreter, these are converted to machine-understandable code.
* JIT compiler is a part of JVM. When the JIT compiler is enabled, the JVM analyzes the method calls in the .class files and compiles them to get more efficient and native code. It also ensures that the prioritized method calls are optimized.
* Once the above step is done, the JVM executes the optimized code directly instead of interpreting the code again. This increases the performance and speed of the execution.



# **Copy constructor**

Copy Constructor is the constructor used when we want to initialize the value to the new object from the old object of the same class.

|  |
| --- |
| **class** **InterviewBit**{  String department;  String service;  InterviewBit(InterviewBit ib){  **this**.departments = ib.departments;  **this**.services = ib.services;  }} |

# **Method Overloading:**

**method overloading** is made possible by introducing different methods in the same class consisting of the same name. Still, all the functions differ in the number or type of parameters. It takes place inside a class and enhances program readability.

|  |
| --- |
| **class** **OverloadingHelp** {  **public** **int** **findarea** (**int** l, **int** b) {  **int** var1;  var1 = l \* b;  **return** var1;  }  **public** **int** **findarea** (**int** l, **int** b, **int** h) {  **int** var2;  var2 = l \* b \* h;  **return** var2;  }  } |

# **Method Overriding:**

**Method overriding** is the concept in which two methods having the same method signature are present in two different classes in which an inheritance relationship is present. A particular method implementation (already present in the base class) is possible for the derived class by using method overriding.

|  |
| --- |
| **class** **HumanBeing** {  **public** **int** **walk** (**int** distance, **int** time) {  **int** speed = distance / time;  **return** speed;  }  }  **class** **Athlete** **extends** **HumanBeing** {  **public** **int** **walk**(**int** distance, **int** time) {  **int** speed = distance / time;  speed = speed \* 2;  **return** speed;  }  } |

**Method overloading** is a compile-time polymorphism in which many methods share the same name but have distinct arguments, signatures, and return types.

**Method overriding** is a runtime polymorphism in which the same method with the same arguments or signature is associated with several classes.

# **Final Keyword:**

* **final variable:**
  + When a variable is declared as final in Java, the value can’t be modified once it has been assigned.
  + If any value has not been assigned to that variable, then it can be assigned only by the constructor of the class.
* **final method:**
  + A method declared as final cannot be overridden by its children's classes.
  + A constructor cannot be marked as final because whenever a class is inherited, the constructors are not inherited. Hence, marking it final doesn't make sense. Java throws a compilation error saying - modifier final not allowed here
* **final class:**
  + No classes can be inherited from the class declared as final. But that final class can extend other classes for its usage.

# **Finally:**

* Purpose: The finally block is used with try-catch blocks to ensure that a specific piece of code executes regardless of whether an exception is thrown or not.
* Usage: The finally block is typically used for cleanup tasks, such as closing files, releasing resources, or resetting variables.

# **Finalize:**

* Purpose: The finalize() method is a method in the Object class that is called by the garbage collector just before an object is reclaimed.
* Usage: The finalize() method is intended for cleanup operations that must be performed before an object is destroyed

|  |
| --- |
| public class MyClass {  @Override  protected void finalize() throws Throwable {  // Cleanup code to be executed before garbage collection  super.finalize();  }  } |

# **Shallow Copy:**

Only the Reference of the object is copied.

changes in one entity are reflected in the other entities

# **Deep Copy:**

 a new memory allocation happens

changes in one entity are not reflected in the other entities

# **objective of garbage collection**

The main objective of this process is to free up the memory space occupied by the unnecessary and unreachable objects during the Java program execution by deleting those unreachable objects.

# **1. Abstract Class**

* **What is it?**
  + An abstract class is a class that cannot be instantiated on its own. It may contain abstract methods (methods without a body) as well as concrete methods (methods with a body).
* **Purpose:**
  + To provide a common base class that other classes can inherit from. It's used when classes share some common behavior, but some methods need to be implemented differently in each subclass.
* **When to Use:**
  + When you want to share code among closely related classes.
  + When you want to provide a default behavior that can be overridden by subclasses.
  + When you expect subclasses to have some methods in common, but you don't want to force all methods to be implemented in every subclass.
* **Example:**

java

abstract class Animal {

abstract void makeSound(); // Abstract method (no body)

void eat() { // Concrete method

System.out.println("Eating...");

}}

class Dog extends Animal {

void makeSound() { // Must implement this method

System.out.println("Bark");}}

# 2. Interface

* **What is it?**
  + An interface is a contract that a class can implement. It only contains method declarations (before Java 8) or default methods (from Java 8 onward), but no instance fields or constructors.
* **Purpose:**
  + To define a set of methods that a class must implement, regardless of where that class is in the inheritance hierarchy. It's used to specify behaviors that can be shared across different classes that may not be related.
* **When to Use:**
  + When you want to define a capability that can be applied to many classes, even if they are not related (e.g., a Flyable interface for birds, planes, etc.).
  + When you want to implement multiple inheritances (since a class can implement multiple interfaces).
  + When you need to define a contract for what a class should do, without worrying about how it does it.
* **Example:**

java

Copy code

interface Flyable {

void fly(); // Abstract method (no body)

}

class Bird implements Flyable {

public void fly() { // Must implement this method

System.out.println("Flying...");

}

}

class Plane implements Flyable {

public void fly() { // Must implement this method

System.out.println("Plane taking off...");

}

}

**Key Differences:**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Methods** | Can have both abstract and concrete methods | Can only have abstract methods (before Java 8); can have default and static methods (Java 8 onwards) |
| **Fields** | Can have instance variables and constants | Can only have constants (static final fields) |
| **Inheritance** | A class can extend only one abstract class | A class can implement multiple interfaces |
| **Constructors** | Can have constructors (used by subclasses) | Cannot have constructors |
| **Use Case** | When classes are closely related and share behavior | When unrelated classes need to share a capability |

**When to Use Which:**

* **Abstract Class:**
  + Use when you have a clear hierarchical relationship (e.g., Animal -> Dog, Cat).
  + Use when you want to provide common behavior that can be shared and overridden by subclasses.
* **Interface:**
  + Use when you want to define a role or capability (e.g., Flyable, Drivable).
  + Use when you need to apply the same set of methods to different, unrelated classes.
  + Use when you want a class to inherit behavior from multiple sources.

**Summary:**

* **Abstract classes** are for inheritance and sharing code among related classes.
* **Interfaces** are for defining a contract or capability that multiple classes, potentially unrelated, can implement.

# **String vs String Builder vs String Buffer**

Use String when you need an immutable string (e.g., constant values).

Use StringBuilder when you need a mutable string in a single-threaded environment for better performance.

Use StringBuffer when you need a mutable string in a multi-threaded environment, where thread safety is important.

 **String:** Unchangeable text.

 **StringBuilder:** Changeable text for one person.

 **StringBuffer:** Changeable text that’s safe for multiple people.

# **Spring:**

[Sprint boot](https://www.interviewbit.com/blog/spring-boot-architecture/) is a Java-based spring framework used for Rapid Application Development (to build stand-alone microservices). It has extra support for auto-configuration and embedded application servers like Tomcat, jetty, etc.

**Features of Spring Boot that make it different?**

* Creates stand-alone spring application with minimal configuration needed.
* It has embedded tomcat, jetty which makes it just code and run the application.
* Provide production-ready features such as metrics, health checks, and externalized configuration.
* Absolutely no requirement for XML configuration.

## **Components:**

* Spring Boot auto-configuration.
* Spring Boot CLI.
* Spring Boot starter POMs.
* Spring Boot Actuators

## **Advantages:**

* Fast deployment
* High scalability
* Container compatibility
* Minimal configuration
* Lower production time
* Increased productivity
* Reduced development time
* Easy monitoring and management of applications

Spring Boot automatically scans all the components included in the project by using @ComponentScan annotation.

add the @ComponentScan annotation for your class file to scan your components added to your project.

## **Spring Boot CLI :**

Spring Boot CLI is a command-line interface that allows you to create a spring-based java application using Groovy.

Example: You don’t need to create getter and setter method or access modifier, return statement. If you use the JDBC template, it automatically loads for you.

## Spring Initializer:

solves the problem of setting up a framework when you are starting a project from scratch.

**Key Differences:**

1. **Response Handling:**
   * @Controller: Returns a view name, and the response is typically an HTML page.
   * @RestController: Returns the data directly, usually in JSON or XML format.
2. **Annotation Behaviour:**
   * @Controller: Requires @ResponseBody on each method to send the response body directly, if needed.
   * @RestController: Automatically applies @ResponseBody to all methods, so no need to add @ResponseBody explicitly.
3. **Typical Use Case:**
   * @Controller: Used in traditional web applications with server-side rendered views.
   * @RestController: Used in RESTful web services or APIs that return data directly to the client.

## **IOC container**

IoC Container is a framework for implementing automatic dependency injection. It manages object creation and its life-time and also injects dependencies into the class.

@GetMapping is only for HTTP GET request while @RequestMapping can be used to process any HTTP methods like POST, PUT, GET and even DELETE.

define both application and Spring boot-related properties into a file called **application.properties**.

## **dependency Injection:**

The process of injecting dependent bean objects into target bean objects is called dependency injection.

## @RequestMapping:

@RequestMapping is a Spring Boot annotation used to map a URL request to a controller method.