Certification:
Java Full Stack Development
Linkcode Technology, 2023-2024

Date:
June 2025

Description:
This document presents consolidated notes and examples covering essential Java backend development concepts, including OOP principles, multithreading, exception handling, JDBC, and real-world project structures. Compiled and refined based on training, self-study (GFG, JavaTpoint), and real interview experiences as part of the Java Full Stack Development certification at Linkcode Technology.

Java Notes Index

Introduction

Good morning Sir or Maam, my name is Tejas Waghmare from rajgurunagar pune. I am currently pursuing my Bachelor of Engineering in Computer Engineering from NBN Sinhgad School Of Engineering, with an aggregate CGPA of 8.11.

In terms of technical skills, I specialize in Java Full Stack Development, including java adv java, JSP, Java Servlet, Spring Boot and Angular, and have hands-on experience with tools like SQL, HTML, and CSS. I've applied these skills to various projects, such as a dynamic Bank Management system using JSP and Servlete and a Todo Task application integrating Spring Boot and Angular.

I recently completed an internship at KasNet Technology, where I managed and deployed Azure cloud solutions, **improving deployment** efficiency. Additionally, I hold an Azure AZ-900 certification.

I am passionate about problem-solving, enjoy working on innovative projects, and am eager to contribute my skills to a challenging and dynamic work environment. I look forward to discussing how my skills and experiences align with the needs of organization.

1. JVM

The **Java Virtual Machine (JVM)** is the engine that drives Java applications. When a Java program is compiled, it generates bytecode, a platform-independent intermediate code. The JVM is responsible for converting this bytecode into machine code specific to the operating system and hardware, enabling Java's "Write Once, Run Anywhere" functionality.

The JVM performs three main tasks: loading, verifying, and executing code.

The **Class Loader** loads the compiled bytecode into memory,

the **Bytecode Verifier** ensures Java’s security and syntax rules, and the

Execution Engine compiles it into machine instructions.

Additionally, the JVM handles **memory management**, allocating space for objects and performing garbage collection to reclaim unused memory.

The **JRE (Java Runtime Environment)** is a software package that provides everything needed to run Java applications.

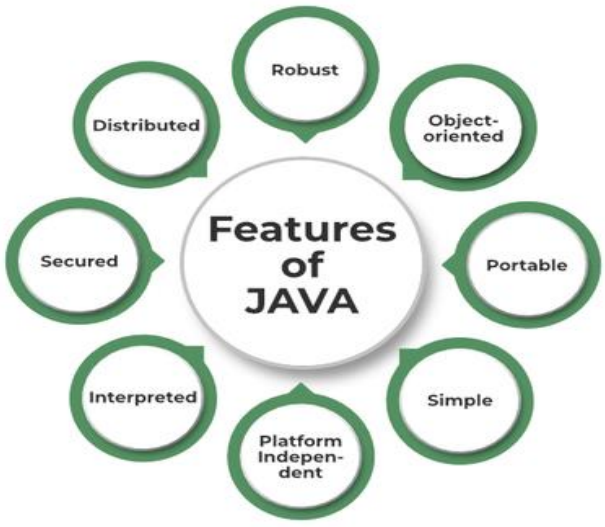
It contains core Java libraries, classes, and resources necessary for program execution, making it a crucial component for end-users who only need to run Java applications.

It includes the **JVM (Java Virtual Machine)**, which executes Java programs, along with the libraries and other components required for the program to function properly.

When you install the JRE on your system, it allows you to run Java programs but does not include tools for writing or compiling Java code.

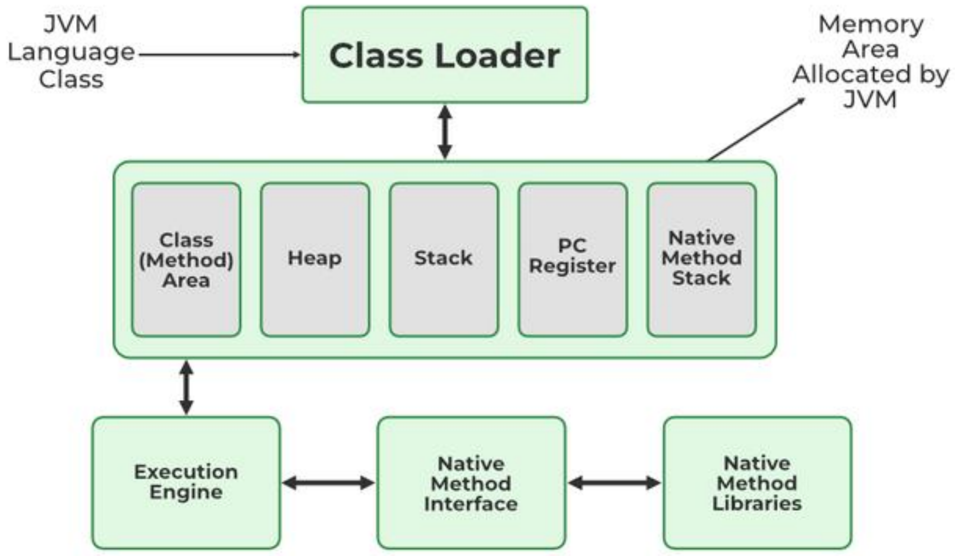
Is Java Platform Independent if then how?

Yes, Java is a Platform Independent language. Unlike many programming languages javac compiles the program to form a bytecode or .class file. This file is independent of the software or hardware running but needs a JVM(Java Virtual Machine) file preinstalled in the operating system for further execution of the bytecode.



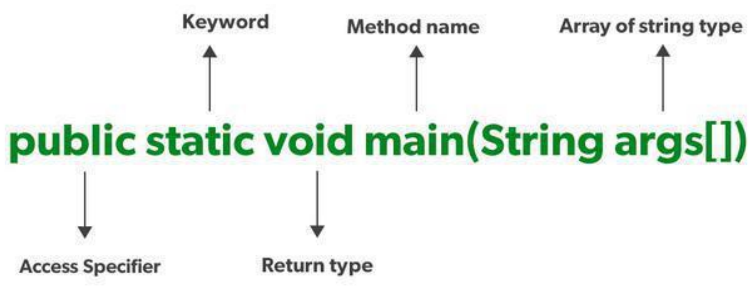
JIT

JIT stands for (Just-in-Time) compiler is a part of JRE(Java Runtime Environment), it is used for better performance of the Java applications during run-time.



JDK:

JDK stands for Java Development Kit which provides the environment to develop and execute Java programs. JDK is a package that includes two things Development Tools to provide an environment to develop your Java programs and, JRE to execute Java programs or applications.



1. **public**: the public is the access modifier responsible for mentioning who can access the element or the method and what is the limit. It is responsible for making the main function globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class.
2. **static**: static is a keyword used so that **we can use the element without initiating** the class so to avoid the unnecessary allocation of the memory.
3. **void**: void is a keyword and is used to specify that a method doesn't return anything. As the main function doesn't return anything we use void.
4. **main**: main represents that the function declared is the main function. It helps JVM to identify that the declared function is the main function.
5. **String args[]**: It stores Java command-line arguments and is an array of type java.lang.String class.

Java String Pool

A Java String Pool is a place in heap memory where all the strings defined in the program are stored.

What will happen if we declare don't declare the main as static?

We can declare the main method without using static and without getting any errors. But, the main method will not be treated as the entry point to the application or the program.

What are Packages in Java?

Packages in Java can be defined as the grouping of related types of classes, interfaces, etc providing access to protection and namespace management.

Explain different data types in Java.

There are 2 types of data types in Java as mentioned below:

1. Primitive Data Type
2. Non-Primitive Data Type or Object Data type

Primitive Data Type: Primitive data are single values with no special capabilities. There are 8 primitive data types:

- **boolean**: stores value true or false
- **byte**: stores an 8-bit signed two's complement integer
- **char**: stores a single 16-bit Unicode character
- **short**: stores a 16-bit signed two's complement integer
- **int**: stores a 32-bit signed two's complement integer
- **long**: stores a 64-bit two's complement integer
- **float**: stores a single-precision 32-bit IEEE 754 floating-point
- **double**: stores a double-precision 64-bit IEEE 754 floating-point

Non-Primitive Data Type: Reference Data types will contain a memory address of the variable's values because it is not able to directly store the values in the memory. Types of Non-Primitive are mentioned below:

- Strings
- Array
- Class
- Object
- Interface

Can we declare Pointer in Java?

No, Java doesn't provide the support of Pointer. As Java needed to be more secure because which feature of the pointer is not provided in Java.

What is the Wrapper class in Java?

Wrapper, in general, is referred to a larger entity that encapsulates a smaller entity. Here in Java, the wrapper class is an object class that encapsulates the primitive data types.

The primitive data types are the ones from which further data types could be created. For example**, integers can further lead to the construction of long, byte, short, etc**. On the other hand, the string cannot, hence it is not primitive.

Why do we need wrapper classes?

The wrapper class is an object class that encapsulates the primitive data types, and we need them for the following reasons:

1. Wrapper classes are final and immutable
2. Provides methods like valueOf(), parseInt(), etc.
3. It provides the feature of autoboxing and unboxing.

Differentiate between instance and local variables.

Instance Variable	**Local Variable**
Declared outside the method, directly invoked by the method.	Declared within the method.
Has a default value.	No default value
It can be used throughout the class.	The scope is limited to the method.

Explain the difference between instance variable and a class variable.

Instance Variable: A class variable **without a static modifier** known as an instance variable is typically **shared by** all instances of the class. These variables can have distinct values among several objects. The contents of an instance variable are completely independent of one object instance from another because they are related to a specific object instance of the class.

Class Variable: Class Variable variable can be declared anywhere at the class level using the keyword static. These variables can only have one value when applied to various objects. These variables can be shared by all class members since they are not connected to any specific object of the class.

static keyword

In Java, the **static keyword** is used to indicate that a variable, method, or block belongs to the **class** rather than any specific object of the class.

This means it can be accessed **without creating an instance of the class.**

Key Points:

1. **Static Variables:**
    - Shared across all objects of the class.
    - Example: A static variable count keeps the same value for all objects.
2. **Static Methods:**
    - Can be called using the class name, without creating an object.
    - Example: Math.pow() is a static method in Java.
    - Cannot access non-static (instance) variables directly.
3. **Static Blocks:**
    - Used for initializing static variables when the class is loaded.
    - Runs only once, when the class is first used.

this keyword

In Java, the **this keyword** refers to the current object of the class.

It is used to work with the **instance variables, methods, or constructors** of that specific object.

Key Uses of this Keyword:

1. **Access Instance Variables:**
   When instance variables and method parameters have the same name, this helps distinguish the instance variable from the local parameter.

2. **Call Another Constructor (Constructor Chaining):**
   this() is used to call one constructor from another within the same class, making code more organized.
3. **Pass the Current Object:**
   this can be passed as an argument to methods or constructors, allowing another method or class to work with the current object.

In simple terms, the this keyword allows an object to refer to itself, which is helpful in **solving naming conflicts, reusing constructors, and implementing advanced object-oriented** techniques.

System.out	**System.err**
It will print to the standard out of the system.	It will print to the standard error.
It is mostly used to display results on the console.	It is mostly used to output error texts.

How many ways you can take input from the console?

There are two methods to take input from the console in Java mentioned below:

1. Using Command line argument
2. Using Buffered Reader Class
3. Using Console Class
4. Using Scanner Class

Difference in the use of print, println, and printf.

print, println, and printf all are used for printing the elements but print prints all the elements and the cursor remains in the same line. println shifts the cursor to next line. And with printf we can use format identifiers too.

Explain the difference between >> and >>> operators.

Operators like >> and >>> seem to be the same but act a bit differently. >> operator shifts the sign bits and the >>> operator is used in shifting out the zero-filled bits.

What is dot operator?

The Dot operator in Java is used to access the instance variables and methods of class objects.

What are the differences between String and StringBuffer?

String	**StringBuffer**
Store of a sequence of characters.	Provides functionality to work with the strings.
It is immutable.	It is mutable (can be modified and other string operations could be performed on them.)
No thread operations in a string.	It is thread-safe (two threads can’t call the methods of StringBuffer simultaneously)

48. What are the differences between StringBuffer and StringBuilder?

StringBuffer	StringBuilder
StringBuffer provides functionality to work with the strings.	StringBuilder is a class used to build a mutable string.
It is thread-safe (two threads can't call the methods of StringBuffer simultaneously)	It is not thread-safe (two threads can call the methods concurrently)
Comparatively slow as it is synchronized.	Being non-synchronized, implementation is faster

OOP (Object-Oriented Programming):

OOP (Object-Oriented Programming) is a programming paradigm based on the concept of **objects**, which represent real-world entities. These objects have two main components: **attributes** (data) and **behavior** (methods or functions).

Key Principles of OOP:

1. **Encapsulation:**
    - Bundling data (variables) and methods (functions) into a single unit (class) and restricting direct access to some components for better security and control.
    - Example: Using private variables with public getter and setter methods.
2. **Inheritance:**
    - A mechanism where one class (child) inherits attributes and behaviors from another class (parent), enabling code reuse and hierarchical relationships.
3. **Polymorphism:**
    - The ability for a single method or object **to take multiple forms**, allowing flexibility in method behavior.
    - Example: Method Overloading (same name, different parameters) and Method Overriding (same name and parameters, but different implementation).

4. **Abstraction:**
    - Hiding implementation details and exposing only essential features through abstract classes or interfaces.
    - Focuses on **what** an object does rather than **how** it does it.

What are Classes in Java?

In Java, Classes are the collection of objects sharing similar characteristics and attributes.

Classes represent the blueprint or template from which objects are created.

Classes are not real-world entities but help us to create objects which are real-world entities.

What is an object?

The object is a real-life entity that has certain properties and methods associated with it.

The object is also defined as the instance of a class.

An object can be declared using a new keyword.

Real-Time Example:

- **Class:** Think of a "Car" as a class.

  It defines properties like color, brand, and model

  behaviors like start, stop, and accelerate.
- **Object:** A specific car, like a "Red Toyota Corolla," is an object.

  It has the actual color, brand, and model values defined in the class.

Constructor:

A **constructor** in Java is a special method used to initialize objects of a class.

It is called automatically when an object is created.

The constructor sets up the initial state of the object by assigning values to its instance variables or performing other setup tasks.

Key Features of a Constructor:

1. **Same Name as Class:**
    - The constructor's name must match the class name.
2. **No Return Type:**
    - Constructors do not have a return type, not even void.
3. **Automatic Invocation:**
    - It is called automatically when an object is created.

Types of Constructors:

1. **Default Constructor:**
    - A no-argument constructor provided by Java if no constructor is explicitly defined.
    - Initializes objects with default values.
2. **Parameterized Constructor:**
    - A constructor that takes arguments to initialize an object with specific values.
3. **Constructor Overloading:**
    - A class can have multiple constructors with different parameters, allowing flexibility in object creation.

In simple terms, a constructor is like a setup process for an object, making sure it's ready to use right after being created.

What happens if you don't provide a constructor in a class?

If you don't provide a constructor in a class in Java, the compiler automatically generates a default constructor with no arguments and no operation which is a default constructor.

What do you understand by copy constructor in Java?

A **copy constructor** in Java is a special type of constructor used to create a new object by copying the values from an existing object.

It essentially duplicates an object by initializing the new object's fields with the same values as another object of the same class.

What are the differences between the constructors and methods?

Java constructors are used for initializing objects. During creation, constructors are called to set attributes for objects apart from this few basic differences between them are:

1. Constructors are only called when the object is created but other methods can be called multiple times during the life of an object.
2. Constructors do not have a return type, whereas methods have a return type, which can be void or any other type.
3. Constructors are used to setting up the initial state but methods are used to perform specific actions.

What is an Interface?

In Java, an **interface** is like a **contract** that defines a set of rules (methods) that a class must follow.

It specifies **what** a class should do, but not **how** it should do it.

Key Points:

- An interface only contains method declarations (no method body).
- A class that implements the interface must provide the implementation for all the methods.
- It helps in achieving abstraction and multiple inheritance.

Example in Simple Terms:

Think of an interface as a **remote control**. The remote has buttons like powerOn, volumeUp, and channelChange. The remote doesn't care whether it's controlling a TV, an AC, or a music system—each device just needs to implement these functions in its own way.

Features of the Interface are mentioned below:

- The interface can help to achieve total abstraction.
- Allows us to use multiple inheritances in Java.
- Any class can implement multiple interfaces even when one class can extend only one class.
- It is also used to achieve loose coupling.

```
interface InterfaceName {
    // Abstract method (no body)
    void method1();

    // Default method (with body, since Java 8)
    default void method2() {
        System.out.println("This is a default method.");
    }
}
```

Abstraction

In Java, **abstraction** is the process of hiding unnecessary details and showing only the essential features of an object or system.

It focuses on **what** an object does, not **how** it does it.

Key Points:

- Abstraction is achieved using **abstract classes** and **interfaces**.
- It helps simplify complex systems by breaking them into smaller, manageable parts.

Example in Simple Terms:

Think of a **coffee machine**.

- You press a button to make coffee (what it does).
- You don't need to know how the machine grinds beans or boils water (how it does it).

In short, abstraction helps you interact with a system without worrying about its internal complexities.

```
abstract class ClassName {
    // Abstract method (no body)
    abstract void abstractMethod();

    // Concrete method (with body)
    void concreteMethod() {
        System.out.println("This is a concrete method.");
    }}
```

Abstract Class	**Interface Class**
Both abstract and non-abstract methods may be found in an abstract class.	The interface contains only abstract methods.
Abstract Class supports Final methods.	The interface class does not support Final methods.
Multiple inheritance is not supported by the Abstract class.	Multiple inheritances is supported by Interface Class.
Abstract Keyword is used to declare Abstract class.	Interface Keyword is used to declare the interface class.
extend keyword is used to extend an Abstract Class.	**implements** keyword is used to implement the interface.
Abstract Class has members like protected, private, etc.	All class members are public by default.

Encapsulation

In Java, **encapsulation** is the process **of bundling data** (variables) and **methods** (functions) that operate on that data into a single unit, typically a class.

It also involves **restricting direct access** to some of the object's components to ensure better control and security.

Key Features:

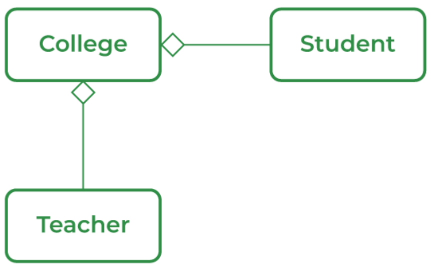
1. **Data Hiding:**
   - By using **private access modifiers** for variables, you hide them from external access.
2. **Controlled Access:**
   - You provide access to the hidden data through **public getter and setter methods**.

Example in Simple Terms:

Think of a **capsule (medicine)**—it encapsulates the ingredients inside a sealed container. Similarly, in Java, a class hides its internal details and only exposes what is necessary through methods.

In short, **encapsulation** protects data from unauthorized access and makes the code more modular, secure, and easy to maintain.

What do you mean by aggregation?



- Aggregation is a term related to **the relationship between two classes** best described as a "**has-a**" relationship.
- This kind is the most specialized version of association.
- It is a unidirectional association means it is a **one-way relationship**.
- It contains the reference to another class and is said to have ownership of that class.

Define Inheritance

In Java, **inheritance** is a mechanism where one class (called the **child class** or **subclass**) can get the properties (fields) and behaviors (methods) of another class (called the **parent class** or **superclass**). It allows code to be reused and establishes a relationship between classes.

Key Points:

1. **Reusability:**
   - The **child class** can use the existing code of the **parent class** instead of writing it again.
2. **Extensibility:**
   - The child class can add its own features or modify the inherited ones.
3. **Syntax:**
   - Use the extends keyword for inheritance.

Example in Simple Terms:

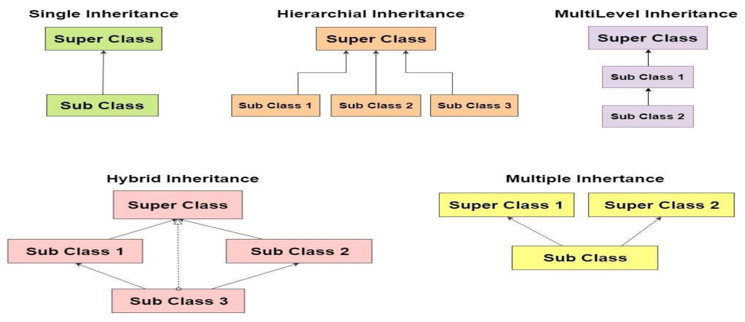
Think of a **family relationship**:

- A child inherits quality like eye color or height from their parents.
- Similarly, in Java, a child class can inherit methods and variables from the parent class.

In short, **inheritance** helps reduce redundancy, makes the code more organized, and establishes a hierarchical relationship between classes.

What are the different types of inheritance in Java?

In Java, Inheritance is of four types:



- **Single Inheritance:**

  When a **child or subclass** extends **only one superclass**, it is known to be single inheritance.

  Single-parent class properties are passed down to the child class.

- **Multilevel Inheritance**

  means a class is **derived** from another class, which **itself** is derived from a **third** class.

  This creates a chain of inheritance where properties and methods are passed down multiple levels.

- **Hierarchical Inheritance:**

  means **multiple child classes** inherit from a **single parent** class.

  The **parent** class serves as a **common base**, and each child class can extend and customize its behavior while sharing the features of the parent class.

- **Multiple Inheritance:**

  When a **one child class** inherits from **multiple parent** classes is known as Multiple Inheritance.

  In Java, it only supports multiple inheritance of interfaces, not classes.

Why we are not using multiple inheritance in Java?

We don't use **multiple inheritance** in Java because it can lead to **ambiguity and complexity**, especially when two parent classes have methods with the same name. This is called the **Diamond Problem**.

Key Reasons:

1. **Ambiguity (Diamond Problem):**
   - If two parent classes have the same method, the child class wouldn't know which one to use, causing confusion.
2. **Complexity:**
   - Handling multiple parent classes makes the code harder to understand and maintain.

3. **Better Alternatives:**
    - Instead of multiple inheritance, Java uses **interfaces** to achieve the same functionality without ambiguity.

Simple Example:

Imagine **Parent1** and **Parent2** both have a method display(). If the child class inherits from both, Java wouldn’t know which display() to execute.

In short, Java avoids multiple inheritance to prevent errors and keep code simpler and more reliable.

What is Polymorphism?

Polymorphism in Java means "one name, many forms."

It allows the same action (method or object) to behave differently based on the context.

Key Points:

1. **Method Overloading (Compile-time Polymorphism):**
    - **Same method** name, but **different parameter** lists in the same class.
    - Example: One method add() can add two numbers or concatenate two strings, depending on the input.
2. **Method Overriding (Runtime Polymorphism):**
    - A child class provides a **specific implementation** for a method that is already defined in its parent class.
    - Child class bap banto… swatach dok lavto.
    - Example: A Dog class overrides the speak() method of the Animal class to bark instead of making a generic animal sound.

Simple Real-Life Example:

Think of a **printer**:

- **Polymorphism in action:** The same print() function prints a photo, a document, or a sticker, depending on the input.

In short, **polymorphism** makes code flexible and allows a single interface to work in multiple ways.

Can we override the static method?

No, as static methods are part of the class rather than the object so we can’t override them.

What is Abstract class?

An **abstract class** in Java is a class that cannot be instantiated (you can't create an object from it directly).

It is used to represent a general concept, providing a base for other classes to build on. An abstract class can have both **abstract methods** (methods without a body) and **concrete methods** (methods with a body).

Key Points:

1. **Abstract Methods:**
    - These methods are declared in the abstract class but don't have a body. The child class must provide the implementation.
2. **Concrete Methods:**
    - These are normal methods with code inside them, and the child class can use them without overriding.
3. **Cannot Instantiate:**
    - You can't create an object of an abstract class directly. Instead, you create objects of its subclasses.

Simple Example:

Think of an **abstract blueprint** for a **vehicle**:

- An abstract class Vehicle might have an abstract method start(), but different vehicles like Car and Bike will implement how the start() method works for them.

In short, an **abstract class** is used to define common characteristics while leaving the specific implementation details to the subclasses.

What is Collection Framework in Java?

Collections are units of objects in Java.

The collection framework is a set of interfaces and classes in Java that are used to represent and manipulate collections of objects in a variety of ways.

The collection framework contains classes(ArrayList, Vector, LinkedList, PriorityQueue, TreeSet) and multiple interfaces (Set, List, Queue, Deque) where every interface is used to store a specific type of data.

✅ Why Use the Collection Framework?

Before Java 1.2, developers relied on arrays and custom data structures. These had limitations like fixed size, no dynamic resizing, and lack of built-in methods for sorting, searching, or manipulation. The Collection Framework solves these by:

- Providing **ready-made, reusable classes and interfaces**
- Ensuring **type safety** using Generics
- Supporting **polymorphism** (one interface, multiple implementations)
- Offering **algorithms** (like sorting, shuffling)
- Allowing **interoperability** between different types of collections

---

🔧 Key Components of the Collection Framework

1. Interfaces

These define the basic types of collections and the operations they support.

Interface	Description
Collection	Root of the collection hierarchy
List	Ordered collection that can contain duplicate elements (ArrayList, LinkedList, Vector)
Set	Unordered collection that cannot contain duplicate elements (HashSet, TreeSet)
Queue	Stores elements in FIFO (First In, First Out) order (PriorityQueue, ArrayDeque)
Deque	Double-ended queue, allows insertion/removal at both ends
Map	Not part of Collection but related; stores key-value pairs (HashMap, TreeMap, LinkedHashMap)

2. Classes

These are concrete implementations of the interfaces.

Class	Description
ArrayList	Dynamic array; fast for random access
LinkedList	Doubly linked list; fast for insertions/deletions
Vector	Synchronized version of ArrayList
HashSet	Stores unique elements, no guaranteed order
TreeSet	Sorted set
HashMap	Key-value storage; allows one null key
TreeMap	Sorted map based on keys

🛠️ Features of Collection Framework

- **Generics**: Type-safe collections (e.g., List<String>)
- **Algorithms**: Built-in methods for sorting, reversing, searching
- **Thread-safe versions**: Collections.synchronizedList(), ConcurrentHashMap
- **Utility classes**: Collections, Arrays

🧠 Important Points for Interviews

1. **Difference between ArrayList and LinkedList**
   - ArrayList uses dynamic array (fast access, slow inserts/deletes)
   - LinkedList uses nodes (slow access, fast inserts/deletes)
2. **Set vs List**
   - List allows duplicates, maintains order
   - Set does not allow duplicates, unordered or sorted (TreeSet)
3. **HashMap vs TreeMap**
   - HashMap is faster, unsorted
   - TreeMap is sorted by key, slower than HashMap
4. **Thread Safety**

- Vector and Hashtable are synchronized (thread-safe)
- ArrayList and HashMap are not thread-safe by default

📘 **Example Code: Using ArrayList**

```
import java.util.*;

public class Demo {
   public static void main(String[] args) {
      List<String> names = new ArrayList<>();
      names.add("Tejas");
      names.add("Sujal");
      names.add("Tejas"); // allowed in List

      for (String name : names) {
         System.out.println(name);
      }
   }
}
```

What is a Vector in Java?

Vectors in Java are similar and can store multiple elements inside them. Vectors follow certain rules mentioned below:

1. Vector can be imported using **Java.util.Vector.**
2. Vector is implemented using a **dynamic array** as the size of the vector increases and decreases depending upon the elements inserted in it.
3. Elements of the Vector **using index numbers**.
4. Vectors are **synchronized in nature** means they only used a **single thread** (only one process is performed at a particular time).

5. **The vector contains many methods that are not part of the collections framework.

What is Exception Handling?

Exception Handling in Java is a way to handle **unexpected events or errors** that occur during the execution of a program.

It ensures the program **doesn't crash** and can handle errors gracefully, allowing it to continue running or **terminate safely**.

Key Points:

1. **What is an Exception?**
    - An exception is an unexpected event, like dividing by zero, accessing an invalid array index, or trying to open a non-existent file.
2. **How it Works:**
    - Java uses a mechanism to detect and handle exceptions using try, catch, finally, and throw blocks.
    - The error is "caught" and managed instead of letting the program crash.

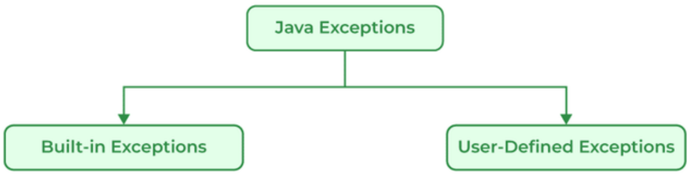
Simple Example:

Imagine you're withdrawing money from an ATM:

- If you try to withdraw more than your balance, the ATM doesn't shut down; instead, it shows an error message like "Insufficient Funds."

In short, **exception handling** makes your program robust and user-friendly by managing errors in a controlled way.

How many types of exceptions can occur in a Java program?



There are generally two types of exceptions in Java:

- **Built-in Exceptions:** Built-in exceptions in Java are provided by the Java Libraries. These exceptions can be further divided into two subcategories

  i.e., checked and unchecked Exceptions.

  Below are some of the built-in exceptions in Java:

  1. ArrayIndexOutOfBoundsExceptions
  2. ClassNotFoundException
  3. FileNotFoundException
  4. IOException
  5. NullPointerException
  6. ArithmeticException
  7. InterruptedException
  8. RuntimeException

- **User-Defined Exceptions:** User-defined exceptions are defined by the programmers themselves **to handle some specific situations** or errors which are not covered by built-in exceptions.

  To define user-defined exceptions a **new class that extends** the appropriate exception class must be defined.

  User-defined Exceptions in Java are used when the built-in exceptions are in Java.

Difference between an Error and an Exception.

Errors	**Exceptions**
Recovering from Errors is not possible.	Recover from exceptions by either using a try-catch block or throwing exceptions back to the caller.
Errors are all unchecked types in Java.	It includes both checked as well as unchecked types that occur.
Errors are mostly caused by the environment in which the program is running.	The program is mostly responsible for causing exceptions.
Errors can occur at compile time as well as run time. Compile Time: Syntax Error, Run Time: Logical Error.	All exceptions occur at runtime but checked exceptions are known to the compiler while unchecked are not.

Explain Runtime Exceptions.

In Java, **Runtime Exceptions** are errors that occur during the program's **execution** (runtime). They are **unchecked** exceptions, meaning the **compiler doesn't force you to handle** them or declare them in the **throws clause**. Examples include:

- **NullPointerException:** When you try to use an object reference that is null.
- **ArrayIndexOutOfBoundsException**: When accessing an invalid index in an array.
- **ArithmeticException**: Like dividing a number by zero.

These exceptions often indicate bugs in the program that you need to fix.

What is NullPointerException?

It is a type of run-time exception that is thrown when the program attempts to use an object reference that has a null value.

The main use of NullPointerException is to indicate that **no value is assigned to a reference variable**, also it is used for implementing data structures like linked lists and trees.

Checked Exception:

Checked Exceptions are the exceptions that **are checked during compile** time of a program. In a program, if some code within a method throws a checked exception, then the method must either handle the exception or must specify the exception using the throws keyword.

Unchecked Exception:

Unchecked are the exceptions that are **not checked at compile time** of a program. Exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.

What is the use of the final keyword?

The final keyword is used to make functions non-virtual. By default, all the functions are virtual so to make it non-virtual we use the final keyword.

Final Variable: Its value cannot be changed once assigned.
Example: final int x = 10;

Final Method: It cannot be overridden by child classes.

Final Class: It cannot be extended/inherited.
Example: final class A { }

helps ensure immutability or prevent modifications.

What purpose do the keywords final, finally, and finalize fulfill?

i). final:

final is a keyword is used with the variable, method, or class so that they can’t be overridden.

ii) finally

The **finally block** in Java is used to execute code **after a try-catch block**, regardless of whether an exception was thrown or not. It is typically used to **clean up resources** like closing files, streams, or database connections.

iii). finalize

It is a **method** that is called **just before deleting/destructing the objects** which are eligible for **Garbage collection** to perform clean-up activity.

Super keyword:

The **super** keyword in Java refers to the **parent class** and is used for the following purposes:

1. **Access Parent Class Constructor**:
   It calls the constructor of the parent class.
   Example:

```
class Parent {

   Parent() { System.out.println("Parent Constructor"); }

}


class Child extends Parent {

   Child() {

      super(); // Calls Parent's constructor

      System.out.println("Child Constructor");

   }

}
```

2. **Access Parent Class Methods**:
   It calls a method in the parent class that is overridden in the child class.
   Example:

```
class Parent {

   void display() { System.out.println("Parent Method"); }

}
```

```
class Child extends Parent {
    void display() {
        super.display(); // Calls Parent's display method
        System.out.println("Child Method");
    }
}
```

3. **Access Parent Class Variables**:
   It accesses a variable in the parent class hidden by a child class variable.
   Example:

```
class Parent {
    int x = 10;
}

class Child extends Parent {
    int x = 20;

    void show() {
        System.out.println(super.x); // Access Parent's x
    }
}
```

The super keyword helps manage inheritance.

What is multitasking?

Multitasking in Java refers to a program’s capacity to carry out several tasks at once.

Threads, which are quick operations contained within a single program, can do this. Executing numerous things at once is known as multitasking.

What do you mean by a Multithreaded program?

A **multithreaded program** in Java is a program that can run multiple tasks (threads) **simultaneously**, allowing better use of CPU resources.

- A **thread** is a lightweight unit of execution.
- Multithreading lets a program perform **multiple operations at the same time**, like **downloading a file while processing user input**.

Example:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread is running");
   }
}


public class Main {
   public static void main(String[] args) {
      MyThread t1 = new MyThread(); // Create a thread
      t1.start(); // Start the thread
   }
}
```

Multithreading improves performance, especially in tasks like games, web servers, or processing large data.

What are the advantages of multithreading?

There are multiple advantages of using multithreading which are as follows:

1. Responsiveness: User Responsiveness increases because multithreading interactive application allows running code even when the section is blocked or executes a lengthy process.
2. Resource Sharing: The process can perform **message passing and shared memory** because of multithreading.
3. Economy: We are able to share memory because of which the processes are economical.
4. Scalability: Multithreading on multiple CPU machines increases parallelism.

5. Better Communication: Thread synchronization functions improves inter-process communication.
6. Utilization of multiprocessor architecture
7. Minimized system resource use

What are the two ways in which Thread can be created?

Multithreading is a Java feature that **allows concurrent execution** of two or more parts of a program for **maximum utilization of the CPU**. In general, threads are small, lightweight processes with separate paths of execution. These threads use shared memory, but they act independently, thus if any one thread fails it does not affect the other threads. There are two ways to create a thread:

- By extending the Thread class
- By implementing a Runnable interface.

By extending the Thread class

We create a class that extends the **java.lang.Thread class**. This class **overrides the run()** method available in the Thread class. A thread begins its life inside run() method.

Syntax:

```
public class MyThread extends Thread {
public void run() {
      // thread code goes here
   }
}
```

By implementing the Runnable interface

We create a new class that implements **java.lang.Runnable** interface and override run() method. Then we instantiate a Thread object and call the start() method on this object.

Syntax:

```
public class MyRunnable implements Runnable {
public void run() {
      // thread code goes here
   }
}
```

What is a thread?

Threads in Java are subprocess with lightweight with the **smallest unit of processes** and also has separate paths of execution.

A thread has its own **program counter, execution stack, and local variables**, but it shares the same memory space with other threads in the same process.

Java provides built-in support for multithreading through the **Runnable interface** and the **Thread class**.

Runnable interface

The **Runnable interface** in Java is used to represent a task that can be executed by a thread. It is part of the java.lang package and has a single method:

Method:

java

Copy code

```
void run();
```

- The **run() method** contains the **code that you want the thread to execute.**

Why use Runnable?

1. It allows our class to extend another class (since Java doesn't support multiple inheritance).
2. It separates the task logic from the thread's execution logic.

Example:

```
class MyTask implements Runnable {
   public void run() {
      System.out.println("Task is running");
   }
}
public class Main {
    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread thread = new Thread(task); // Pass task to the thread
        thread.start(); // Start the thread
    }
}
```

Key Point:

Using the Runnable interface is preferred in scenarios where your class already extends another class.

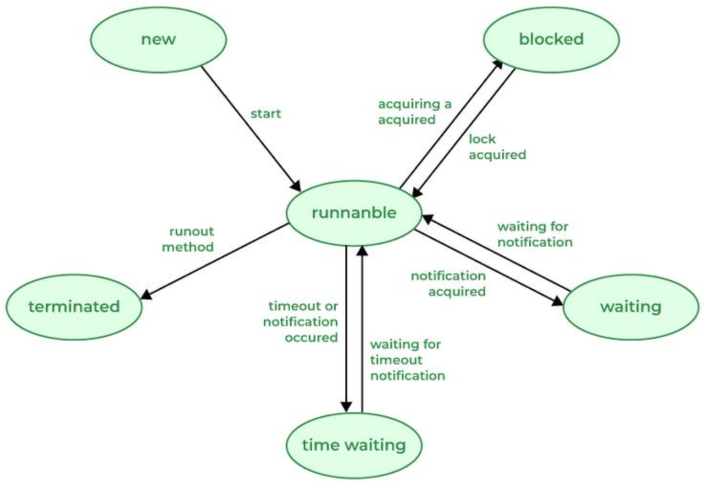
Differentiate between process and thread?

A process and a thread are both units of execution in a computer system, but they are different in several ways:

Process	**Thread**
A process is a program in execution.	A thread is a single sequence of instructions within a process.
The process takes more time to terminate.	The thread takes less time to terminate.
The process takes more time for context switching.	The thread takes less time for context switching.
The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.

Process	Thread
The process is isolated.	Threads share memory.
The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.
The process does not share data with each other.	Threads share data with each other.

Describe the life cycle of the thread? (6 states)



A thread in Java at any point in time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. **New:** The thread has been created but has not yet started.
2. **Runnable:** The thread is running, executing its task, or is ready to run if there are no other higher-priority threads.
3. **Blocked:** The thread is temporarily suspended, waiting for a resource or an event.

4. **Waiting:** The thread is waiting for another thread to perform a task or for a specified amount of time to elapse.
5. **Terminated:** The thread has completed its task or been terminated by another thread.

Explain suspend() method under the Thread class.

The suspend() method of the Thread class in Java **temporarily suspends the execution** of a thread.

When a thread is suspended it goes into a blocked state and it would **not be scheduled by the operating system** which means that it will not be able to execute its task until it is resumed.

There are **more safer and flexible alternatives** to the suspend() methods in the modern java programming language. This method does not return any value.

Syntax:

public final void suspend();

Explain the main thread under Thread class execution.

Java provides built-in support for multithreaded programming.

The main thread is considered the parent thread of all the other threads that are created during the program execution.

The main thread is automatically created when the program starts running.

Why Garbage Collection is necessary in Java?

For Java, Garbage collection is necessary **to avoid memory leaks** which can cause the program to crash and become unstable.

There is no way to avoid garbage collection in Java.

Unlike C++, Garbage collection in Java helps programmers to **focus on the development** of the application **instead of managing memory resources** and worrying about memory leakage.

Java Virtual Machine (**JVM**) automatically **manages the memory periodically by running a garbage collector** which frees up the unused memory in the application.

Garbage collection makes Java **memory efficient** because **it removes unreferenced objects from the heap memory**.

What is JDBC?

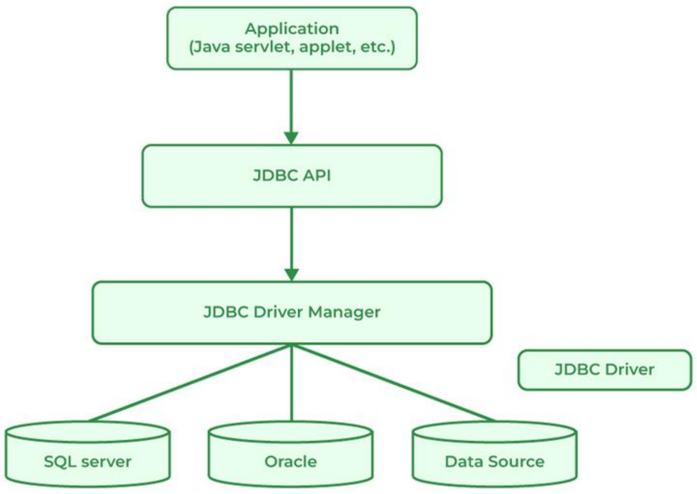
JDBC **standard API** is used to link Java applications and relational databases.

It provides a **collection of classes and interfaces** that let programmers to use the Java programming language to communicate with the database.

The classes and interface of JDBC allow the application to **send requests** which are made by users to the specified database.

There are generally **four components** of JDBC by which it interacts with the database:

1. JDBC API
2. JDBC Driver manager
3. JDBC Test Suite
4. JDBC-ODBC Bridge Drivers



What is JDBC Driver?

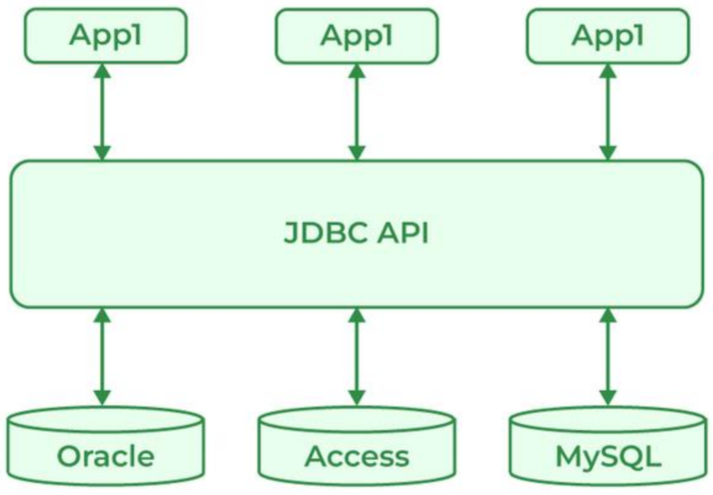
JDBC Driver is a **software component** that is used to **enable** a Java application to interact with the database.

JDBC provides the **implementation of the JDBC API** for a specific database management system,

which allows it to **connect the database, execute SQL statements and retrieve data**.

There are four types of JDBC drivers:

- JDBC-ODBC Bridge driver
- Native-API driver
- Network Protocol driver
- Thin driver



What are the steps to connect to the database in Java?

There are certain steps to connect the database and Java Program as mentioned below:

1. **Import** the Packages
2. Load the drivers using the **forName()** method
3. Register the drivers using **DriverManager**
4. **Establish a connection** using the **Connection class** object
5. Create a statement
6. Execute the query
7. Close the connections

What are the JDBC API components?

JDBC API components provide various methods and interfaces for easy communication with the databases also it provides packages like java Se and java EE which provides the capability of write once run anywhere (WORA).

Syntax:

java.sql.*;

What is JDBC Connection interface?

Java database connectivity interface (JDBC) is a software component that allows Java applications to interact with databases.

To enhance the connection, JDBC requires drivers for each database.

What does the JDBC ResultSet interface?

JDBC ResultSet interface is used to **store the data from the database** and use it in our Java Program.

We can also use ResultSet to **update the data** using updateXXX() methods.

ResultSet object **points the cursor before the first row** of the result data.

Using the next() method, we can iterate through the ResultSet.

What is the JDBC Rowset?

A JDBC RowSet provides a way to **store the data in tabular form**.

RowSet is **an interface** in java that can be used within **the java.sql package**.

The connection between the **RowSet object and the data source** is **maintained throughout its life cycle.**

RowSets are classified into five categories based on implementation mentioned below:

1. JdbcRowSet
2. CachedRowSet
3. WebRowSet
4. FilteredRowSet
5. JoinRowSet

6. **What is the role of the JDBC DriverManager class?**

JDBC DriverManager class **acts as an interface for users and Drivers**. It is used in many ways as mentioned below:

1. It is used to create a **connection between** a Java application and the database.
2. Helps to **keep track of the drivers** that are available.
3. It can help to **establish a connection** between a database and the appropriate drivers.
4. It contains all the methods that can register and deregister the database driver classes.
5. DriverManager.registerDriver() method can maintain the list of Driver classes that have registered themselves.

Difference between == and equals to in java?

In Java:

- **==**: Compares **memory addresses** (references) of two objects. It checks if both variables point to the exact same object in memory.

Example:

```
String a = new String("hello");

String b = new String("hello");

System.out.println(a == b);  // false (different objects in memory)
```

- **equals()**: Compares the **content** of two objects. For example, in the case of strings, it checks if the characters in the strings are the same, even if they are different objects in memory.

Example:

```
String a = new String("hello");

String b = new String("hello");

System.out.println(a.equals(b));  // true (same content)
```

In short:

- **==** checks if two variables refer to the same object.
- **equals()** checks if two objects have the same content.

1. Requirement (What problem is this project solving?)

The project solves the problem of managing core banking operations online. It provides a convenient, secure, and user-friendly way for customers to perform essential banking tasks like checking balance, depositing money, withdrawing cash, and mobile recharge, eliminating the need to visit a bank physically.

---

2. Why did you select this project?

I chose this project because online banking has become a necessity in today's world, and I wanted to understand how to build a secure, efficient, and interactive system that solves real-world problems. It also gave me a chance to apply my knowledge of web development, databases, and security.

---

3. Basic Structure

The project is structured into three main layers:

- **Frontend**: A user-friendly interface for customers to interact with banking features.
- **Backend**: Logic implemented using JSP and Servlet for processing user requests.
- **Database**: JDBC is used to manage and retrieve data securely from the database for transactions.

---

4. Files, Folders, and Libraries

- **Files/Folders**:

    - JSP files for the frontend.
    - Servlet files for handling logic.
    - Config files for database connections.
- **Libraries**:
    - JDBC for database connectivity.
    - Java libraries for handling authentication and transaction logic.

---

5. Workflow

1. The user logs into the system using secure authentication.
2. They navigate to the desired banking feature (e.g., check balance, deposit, withdraw, mobile recharge).
3. The system sends their request to the backend via Servlets.
4. The backend processes the request, interacts with the database via JDBC, and updates/returns the required information.
5. The user receives the output on the frontend, such as a balance update or confirmation message.
