

1. What is the difference between JDK, JRE, and JVM?

Aspect	JDK (Java Development Kit)	JRE (Java Runtime Environment)	JVM (Java Virtual Machine)
Full Form	Java Development Kit	Java Runtime Environment	Java Virtual Machine
Components	Includes tools for developing Java applications, compiler, debugger, and documentation	Provides runtime environment to execute Java applications, libraries, and JVM components	Virtual machine responsible for executing Java bytecode
Contains	Compiler, debugger, Java libraries, and documentation	Java libraries, JVM, and necessary runtime components	Interpreter and runtime system that executes Java bytecode
Use	Needed for Java application development	Required to run Java applications	Executes Java bytecode and manages memory and resources
Development	Used by developers for writing, compiling, and debugging Java code	Not required for development but needed to execute Java applications	Not used for development but essential for executing Java applications

2. Why java is platform independent?

Java is considered platform-independent because of its "write once, run anywhere" capability. This means that you can write Java code on one operating system and run it on any other operating system without modification. Here's how it works:

1. Java Source Code: You write your Java program in a text file using the Java programming language. This is called the source code.

2. Java Compiler: The source code is then compiled by the Java compiler (javac). Instead of translating the source code directly into machine code (which is specific to an operating system), the Java compiler translates it into an intermediate form called bytecode. This bytecode is stored in a file with a `.class` extension.

3. Java Bytecode: The bytecode is not specific to any particular type of computer or operating system. It's a universal code that can be understood by any Java Virtual Machine (JVM).

4. Java Virtual Machine (JVM): The JVM is a special program that runs on your computer. There are different JVMs for different operating systems (Windows, Mac, Linux, etc.), but they all understand and can execute Java bytecode.

When you run a Java program, the JVM interprets the bytecode and translates it into machine code that your computer's operating system can understand. Since the JVM handles the platform-specific details, the same Java bytecode can run on any operating system that has a compatible JVM.

In summary:

- Write once: You write and compile your Java program into bytecode.
- Run anywhere: Any computer with a JVM can run your bytecode, regardless of the underlying operating system.

Java is Platform Independent and **JVM** Is platform dependent.

Code Bashers

3. What is the difference between String, StringBuilder, and StringBuffer?

Aspect	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Modifications	Creating a new instance for each modification, inefficient for frequent modifications	Efficient for frequent modifications, modifies the same instance	Efficient for frequent modifications, modifies the same instance
Thread Safety	Thread-safe (immutable), safe for concurrent use	Not thread-safe, not safe for concurrent use	Thread-safe, safe for concurrent use
Performance	Less efficient for concatenation and modifications, creates a new instance for each operation	More efficient for concatenation and modifications, modifies the same instance	More efficient for concatenation and modifications, modifies the same instance
Usage	Suitable for situations where the content is constant or rarely changed	Suitable for situations where frequent modifications are required and thread safety is not a concern	Suitable for situations where frequent modifications are required and thread safety is a concern

4. What is string Pool ?

The string pool is a special memory area in Java where string literals are stored. When you create a string literal (e.g., `String s = "hello";`), Java checks the string pool first. If the string already exists, it reuses the existing string. If not, it adds the new string to the pool. This helps save memory because identical strings are shared rather than duplicated.

java

```
String str1 = "hello";
String str2 = "hello";
String str3 = new String("hello");

// str1 and str2 point to the same string in the string pool
System.out.println(str1 == str2); // Output: true

// str3 is a new object, not from the string pool
System.out.println(str1 == str3); // Output: false

// str3 can be interned to make it point to the string pool
str3 = str3.intern();
System.out.println(str1 == str3); // Output: true
```

5. Explain Constructor and Types of Constructor

Constructor in Java

A constructor in Java is a special method used to initialize objects. It is called when an instance of a class is created. The constructor has the same name as the class and does not have a return type.

Types of Constructors

1. Default Constructor

- A constructor with no parameters.
- If no constructor is defined in a class, Java provides a default constructor automatically.
- Used to initialize objects with default values.

2. Parameterized Constructor

- A constructor that takes one or more parameters.
- Allows you to provide different values while creating an object.
- Used to initialize objects with custom values.

These constructors help ensure that objects are properly initialized before they are used in your program.

```
public class MyClass {  
    int value;  
  
    // Default constructor  
    MyClass() {  
        value = 0;  
    }  
  
    // Parameterized constructor  
    MyClass(int val) {  
        value = val;  
    }  
  
    public static void main(String[] args) {  
        MyClass obj1 = new MyClass(); // Calls default constructor  
        MyClass obj2 = new MyClass(10); // Calls parameterized constructor  
  
        System.out.println("obj1 value: " + obj1.value); // Output: 0  
        System.out.println("obj2 value: " + obj2.value); // Output: 10  
    }  
}
```

6. Explain OOPS Concepts in Java

Encapsulation : Encapsulation means binding of data (attributes) and methods (behaviors) that operate on the data into a single unit, called a class. This helps keep the implementation details hidden from the outside world, allowing for better organization and security.

By encapsulation we can keep data related to two different Department separate. For example in a company there are two department Sales and Accounts. So here sales data and people are separate from Accounts data and people.

Abstraction : Abstraction means showing only the necessary details to user and hiding the implementation of it.

For example : We use ATM , we just press some buttons and cash comes out , but we dont know what actually is happening behind machine.

Polymorphism : It means existing in various forms . Two important types of Polymorphism is discussed later in this pdf.

Inheritance : Inheritance is a way to create new classes based on existing ones. It allows a new class (called a derived class or subclass) to inherit properties and behaviors from an existing class (called a base class or superclass). This promotes code reuse and helps establish a hierarchy of classes.

7. What are Interfaces in Java ? Why static and default methods were introduced in Java 8

Interfaces in Java provide a way to define a contract for classes. They contain only method signatures (without implementations) and constants. Classes that implement an interface must provide implementations for all the methods declared in the interface.

We use interfaces to achieve abstraction and provide a common way for classes to interact without worrying about the specific implementation details. They allow us to define a set of methods that classes must implement, ensuring consistency and interoperability in our code.

Static and default methods were introduced in Java 8 to enhance the functionality of interfaces:

Static methods in interfaces allow us to define methods that belong to the interface itself, rather than to individual implementing classes. These methods can be called directly on the interface without the need for an implementing class instance.

Default methods in interfaces provide a way to add new methods to existing interfaces without breaking compatibility with implementing classes. These methods have a default implementation in the interface itself, which can be overridden by implementing classes if needed.

```
// Interface definition
interface Vehicle {
    // Abstract method (method signature only)
    void drive();

    // Default method with a default implementation
    default void stop() {
        System.out.println("Vehicle stopped");
    }

    // Static method in interface
    static void honk() {
        System.out.println("Beep beep!");
    }
}
```

8. What is the difference between abstract class and interface in Java?

Aspect	Abstract Class	Interface
Purpose	Used to define common behavior and characteristics for subclasses	Used to define a contract for implementing classes
Method Implementation	Can have both abstract and concrete methods	Can only have abstract methods
Constructor	Can have constructors	Cannot have constructors
Multiple Inheritance	Does not support multiple inheritance	Supports multiple inheritance
Access Modifiers	Can have access modifiers for methods and variables	Methods are implicitly public and abstract, variables are implicitly public, static, and final
Extends vs. Implements	Subclasses extend abstract classes	Classes implement interfaces
Example	<pre>`java abstract class Shape { ... }`</pre>	<pre>`java interface Drawable { ... }`</pre>

9. Explain Different Types of Interfaces

Marker Interface:

A Marker Interface, also known as a tag interface, is an interface that doesn't declare any methods. It simply marks (or tags) a class to provide some information to the compiler or runtime environment.

Classes that implement marker interfaces indicate that they possess certain characteristics or capabilities.

Examples of marker interfaces in Java include `Serializable`, `Cloneable`, and `Remote`.

java

```
// Marker Interface
interface Serializable {
    // Empty interface, doesn't declare any methods
}

// Class implementing marker interface
class MyClass implements Serializable {
    // Class body
}
```

Functional Interface:

A Functional Interface is an interface that contains only one abstract method (SAM), though it can have multiple default or static methods. Functional Interfaces are used to enable functional programming in Java

Java 8 introduced the `@FunctionalInterface` annotation to explicitly mark an interface as a functional interface.

```
java

// Functional Interface
@FunctionalInterface
interface MyFunctionalInterface {
    void myMethod(); // Single abstract method
    // It can contain default and static methods as well
}

// Using lambda expression to implement functional interface
public class Main {
    public static void main(String[] args) {
        MyFunctionalInterface obj = () -> System.out.println("Executing myMethod");
        obj.myMethod(); // Output: Executing myMethod
    }
}
```

10. What are Access Modifiers in Java?

1. **Private:** The `private` access modifier restricts access to members only within the same class. They are not accessible outside the class, not even in subclasses.
2. **Default (Package-private):** If no access modifier is specified, it is considered as default access. Members with default access are accessible only within the same package. They are not accessible from outside the package.
3. **Protected:** The `protected` access modifier allows access to members within the same package and subclasses (even if they are in a different package).
4. **Public:** The `public` access modifier allows unrestricted access to members from any other class or package.

```
public class PublicClass {  
    public int publicNumber = 10; // Public variable  
    private int privateNumber = 20; // Private variable  
    protected int protectedNumber = 30; // Protected variable  
    int defaultNumber = 40; // Default variable  
}
```

11. Difference Bw Method Overloading and Method Overriding

Feature	Method Overloading	Method Overriding
Definition	Having multiple methods with the same name but different parameters within the same class	Redefining a method in a subclass that is already defined in the superclass with the same parameters
Purpose	Increases the readability of the program and provides flexibility by allowing methods to handle different data types or a different number of parameters	Provides specific implementation of a method that is already defined in the superclass to achieve runtime polymorphism
Parameters	Must have different parameters (type, number, or both)	Must have the same parameters (same type and order)
Return Type	Can have the same or different return type	Must have the same return type (or covariant return type in Java 5 and above)
Access Modifier	Can have any access modifier	Access level cannot be more restrictive than the overridden method in the superclass

Overload example

```
java

class OverloadExample {
    void display(int a) {
        System.out.println("Argument: " + a);
    }

    void display(double a) {
        System.out.println("Argument: " + a);
    }

    void display(int a, int b) {
        System.out.println("Arguments: " + a + " and " + b);
    }
}
```

Override Example

```
java

class Superclass {
    void display() {
        System.out.println("Display method in Superclass");
    }
}

class Subclass extends Superclass {
    @Override
    void display() {
        System.out.println("Display method in Subclass");
    }
}
```


12. Explain Final , static and super Keyword In Java`

Final Keyword

Final Variable: A variable declared as final cannot be changed once it is assigned.

Final Method: A method declared as final cannot be overridden by subclasses.

Final Class: A class declared as final cannot be subclassed.
static Keyword

Static Keyword

Static Variable : A variable declared as static belongs to the class rather than instances of the class.

Static Method: A method declared as static belongs to the class and can be called without creating an instance of the class.

Static Block: A block of code that is used to initialize static variables and is executed when the class is loaded.

Super Keyword

Access Superclass Method: The super keyword is used to call a method from the superclass.

Access Superclass Variable: The super keyword is used to access a variable from the superclass.

Call Superclass Constructor: The super keyword is used to call a constructor from the superclass.

```
class Parent {  
    final int finalNumber = 10; // Final variable  
  
    static String staticString = "Static String"; // Static variable  
  
    void display() {  
        System.out.println("Display method in Parent class");  
    }  
}  
  
class Child extends Parent {  
    void accessSuper() {  
        System.out.println(super.finalNumber); // Access final variable from superclass  
        super.display(); // Call method from superclass  
    }  
  
    static void displayStatic() {  
        System.out.println(staticString); // Access static variable within the same class  
    }  
}
```

13. Explain Concept of Multithreading in java

Definition:

Multithreading is a programming technique that allows multiple threads to run concurrently within a single program. Each thread runs a separate path of execution, enabling the program to perform multiple tasks simultaneously.

Key Concepts:

Thread: A thread is a lightweight subprocess, the smallest unit of processing. It has its own call stack but shares resources with other threads in the same process.

Main Thread: When a Java application starts, the JVM creates a main thread to execute the main method.

Creating Threads: In Java, threads can be created in two main ways:
By extending the Thread class.
By implementing the Runnable interface.

Advantages:

Concurrency: Multiple threads can run concurrently, making efficient use of CPU resources.

Improved Performance: Multithreading can improve the performance of applications by allowing tasks to run in parallel, reducing execution time.

Responsiveness: In GUI applications, multithreading keeps the user interface responsive by performing time-consuming tasks in the background.

java

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Thread is running.");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        MyThread thread = new MyThread(); // Creating a new thread  
        thread.start(); // Starting the thread  
    }  
}
```

14. Explain synchronised Keyword in java

The **synchronized** keyword in Java is used to control access to a block of code or a method by multiple threads. It ensures that only one thread can execute the synchronized block or method at a time, which helps to prevent thread interference and consistency problems.

Synchronized Method: When a method is declared as **synchronized**, only one thread can execute that method on the same object instance at any given time.

Synchronized Block: You can also synchronize a specific block of code inside a method. This gives more control and can reduce the overhead of synchronization by limiting it to only the critical section of the code.

java

```
public class BankAccount {  
    private int balance = 0;  
  
    public synchronized void deposit(int amount) {  
        balance += amount;  
    }  
  
    public synchronized int getBalance() {  
        return balance;  
    }  
}
```

15. What are errors and Exceptions?

Aspect	Error	Exception
Cause	Typically caused by environment or programming mistakes	Generally caused by code issues during execution
Handling	Usually cannot be handled gracefully; indicates serious issues	Can be caught and handled in code
Examples	Out-of-memory error, stack overflow	NullPointerException, IndexError

16. What are wrapper classes?

Wrapper classes in Java are used to convert primitive data types into objects. In Java, primitive types like `int`, `float`, `char`, etc., are not objects. Wrapper classes like `Integer`, `Float`, `Character`, etc., provide a way to treat primitive types as objects. This allows you to use them in places where objects are required, like in collections (e.g., `ArrayList`) or when working with Java generics.

```
java

public class WrapperExample {
    public static void main(String[] args) {
        // Primitive data type
        int primitiveInt = 10;

        // Using a wrapper class
        Integer wrapperInt = Integer.valueOf(primitiveInt);

        System.out.println("Primitive int: " + primitiveInt);
        System.out.println("Wrapper Integer: " + wrapperInt);
    }
}
```

Output for both is 10

17. Explain Optional Classes In Java

Definition:

The Optional class in Java is a container object which may or may not contain a non-null value. It is used to avoid null checks and prevent NullPointerException.

Purpose:

Avoid NullPointerException: Provides a safer way to handle potentially null values.

Express Intention: Clearly indicates that a value might be absent.

Basic Methods:

Common Methods

isPresent(): Checks if a value is present.

ifPresent(Consumer): Executes a code block if a value is present.

orElse(T other): Returns the value if present, otherwise returns the default value.

orElseGet(Supplier): Returns the value if present, otherwise invokes a supplier function.

orElseThrow(Supplier): Returns the value if present, otherwise throws an exception.

```
public class Main {  
    public static void main(String[] args) {  
        String str = "Hello, Optional!";  
  
        // Create an Optional with a non-null value  
        Optional<String> optional = Optional.of(str);  
  
        // Check if the optional value is present  
        if (optional.isPresent()) {  
            System.out.println("Value is present: " + optional.get());  
        }  
  
        // Perform an action if the optional value is present  
        optional.ifPresent(value -> System.out.println("Value is present: " + value));  
  
        // Provide a default value if the optional value is absent  
        String defaultValue = optional.orElse("Default Value");  
        System.out.println("Value or default: " + defaultValue);  
  
        // Transform the optional value using map  
        optional.map(String::toUpperCase)  
            .ifPresent(value -> System.out.println("Transformed value: " + value));  
    }  
}
```


18. Explain Stream Apis in Java

Definition:

Stream APIs in Java provide a modern way to process collections of objects. They allow you to perform operations like filtering, mapping, and reducing in a more readable and concise way.

Key Features:

Declarative: Allows you to write concise and readable code.

Chained Operations: Supports method chaining for operations.

Lazy Evaluation: Operations are not executed until a terminal operation is called.

Common Operations:

filter: Selects elements based on a condition.

map: Transforms elements.

collect: Converts the stream back to a collection.

forEach: Iterates over elements.


```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Alice", "Bob", "Mary", "David");

        // Filter names starting with 'A'
        List<String> filteredNames = names.stream()
            .filter(name -> name.startsWith("A"))
            .collect(Collectors.toList());

        // Map names to uppercase
        List<String> uppercaseNames = names.stream()
            .map(String::toUpperCase)
            .collect(Collectors.toList());

        System.out.println("Filtered Names: " + filteredNames); // Output: [Alice]
        System.out.println("Uppercase Names: " + uppercaseNames); // Output: [JOHN, ALICE, BOB, MARY, DAVID]
    }
}
```



19. Explain Lambda Function In Java

Lambda functions in Java are a way to define and pass around blocks of code. They are anonymous functions that can be used as method arguments or to create concise implementations of functional interfaces. Here's a simple explanation with an example:

java

```
// Using lambda expression
interface MyInterface {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        // Lambda expression
        MyInterface myInterface = () -> System.out.println("Hello!");
        myInterface.sayHello();
    }
}
```

20. Date Time Api In Java

The Date Time API in Java, introduced in Java 8, provides classes to handle dates, times, and durations.

Key Points:

The Date Time API provides immutable and thread-safe classes for date and time manipulation.

It supports operations such as addition, subtraction, comparison, formatting, and parsing of dates and times.

```
public class Main {  
    public static void main(String[] args) {  
        // LocalDate: Represents a date (year, month, day)  
        LocalDate currentDate = LocalDate.now();  
        System.out.println("Current Date: " + currentDate);  
  
        // LocalTime: Represents a time (hour, minute, second)  
        LocalTime currentTime = LocalTime.now();  
        System.out.println("Current Time: " + currentTime);  
  
        // LocalDateTime: Represents a date-time (combination of date and time)  
        LocalDateTime currentDateTime = LocalDateTime.now();  
        System.out.println("Current Date-Time: " + currentDateTime);  
    }  
}
```

21. Diff bw Comparator and Comparable

Aspect	Comparator	Comparable
Interface	Separate interface (<code>java.util.Comparator</code>)	Interface implemented by the class itself
Purpose	Used to define custom sorting order for objects	Used to define the natural ordering of objects
Implementation	Implemented in a separate class	Implemented within the class itself
Multiple Criteria	Allows sorting based on multiple criteria	Supports sorting based on a single criterion
Flexibility	Provides flexibility to sort objects in different ways	Provides a fixed sorting order

22. Explain Internal Working of Hashmap

Overview:

HashMap is a data structure that stores key-value pairs and provides constant-time performance for basic operations like get and put.

It uses an array of buckets to store the key-value pairs and a hashing mechanism to determine the index of each key-value pair in the array.

Hashing Mechanism:

When you put a key-value pair into a HashMap, the key is hashed to generate a hash code.

The hash code is then mapped to an index in the array using a hashing function.

This index is the location where the key-value pair will be stored.

Handling Collisions:

Collisions occur when two different keys hash to the same index.

To handle collisions, HashMap uses a linked list (or a tree in Java 8 and later versions) at each bucket to store multiple key-value pairs that hash to the same index.

```
public class Main {  
    public static void main(String[] args) {  
        // Creating a HashMap  
        HashMap<String, Integer> hashMap = new HashMap<>();  
  
        // Putting key-value pairs into the HashMap  
        hashMap.put("apple", 10);  
        hashMap.put("banana", 20);  
        hashMap.put("orange", 30);  
  
        // Retrieving values from the HashMap  
        System.out.println("Value for key 'apple': " + hashMap.get("apple"));  
        System.out.println("Value for key 'banana': " + hashMap.get("banana"));  
        System.out.println("Value for key 'orange': " + hashMap.get("orange"));  
    }  
}
```

23. What is TreeMap and LinkedHashMap

- **TreeMap:**
 - Stores entries in a sorted order based on keys.
 - Uses a red-black tree internally.
 - Does not allow null keys but allows multiple null values.
 - Slower than `HashMap` for insertion/deletion but maintains sorted order.
 - More memory efficient for large datasets due to its balanced tree structure.
- **LinkedHashMap:**
 - Maintains insertion order of keys.
 - Uses a doubly linked list to maintain order along with a hash table.
 - Allows one null key and multiple null values.
 - Faster than `TreeMap` for insertion/deletion operations, similar to `HashMap`.
 - Slightly less memory efficient than `HashMap` due to maintaining insertion order with linked nodes.

24. Explain Collection Framework in Java

Definition:

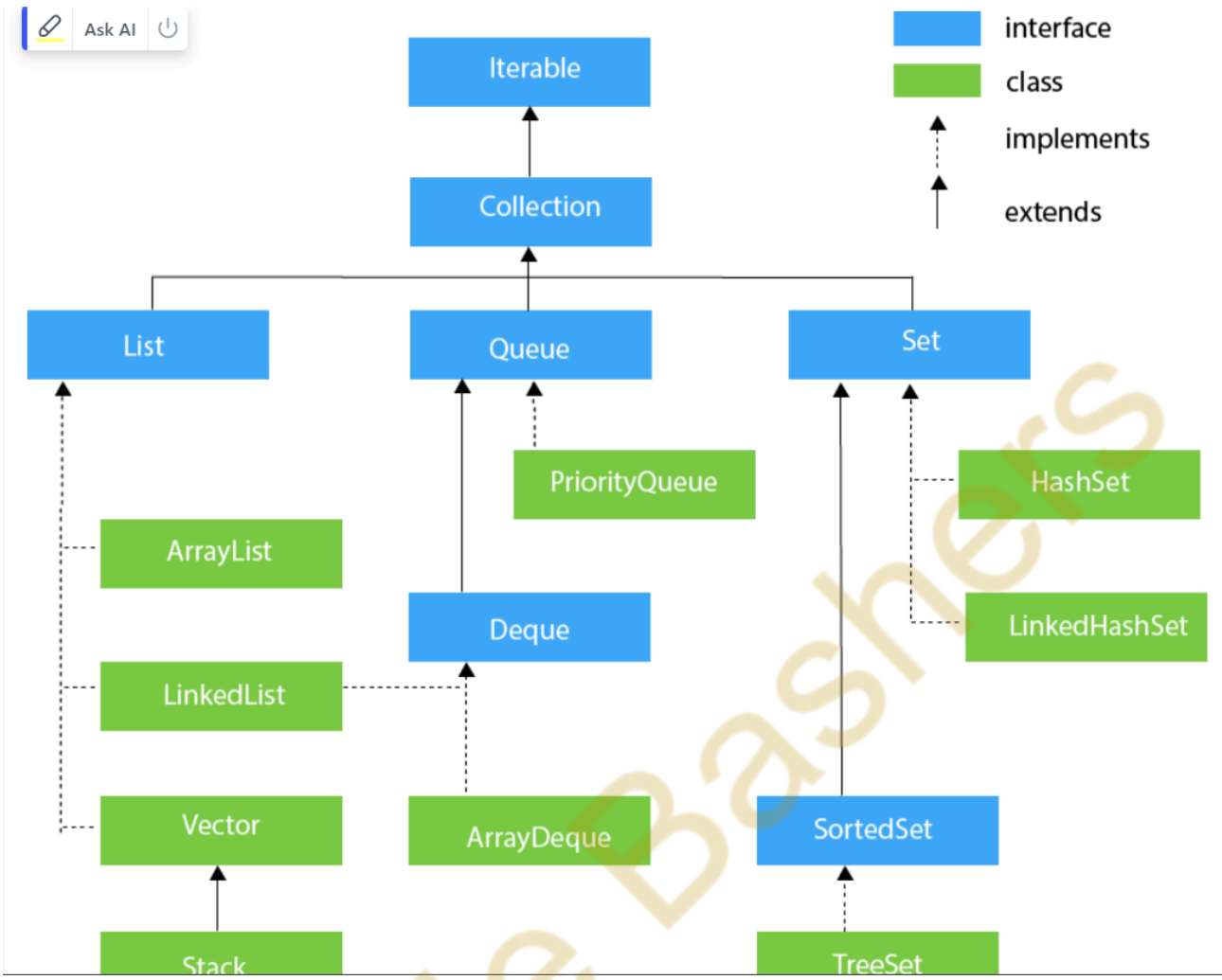
The Collection Framework in Java is a unified architecture for representing and manipulating collections of objects. It consists of interfaces, implementations, and algorithms to work with collections efficiently.

Key Components:

Interfaces: Core interfaces like List, Set, Queue, and Map define common behavior for different types of collections.

Implementations: Concrete classes like ArrayList, HashSet, LinkedList, and HashMap provide specific implementations of the collection interfaces.

Algorithms: The framework includes algorithms for searching, sorting, and manipulating collections, making it easy to perform common tasks.



25. Difference Bw ArrayList and Linked List

Feature	ArrayList	LinkedList
Implementation	Backed by a dynamic array (resizable array)	Doubly linked list
Memory Usage	More memory efficient for storing large lists	Less memory efficient due to additional pointers
Random Access	Efficient for random access (get and set operations)	Less efficient for random access due to traversal
Insertion/Deletion	Less efficient for frequent insertions/deletions	More efficient for frequent insertions/deletions
Performance	Fast for retrieving elements by index	Fast for adding/removing elements from the beginning/end
Iteration	Slower for iteration due to linear access	Faster for iteration due to node traversal
Search Time	$O(1)$ for accessing elements by index	$O(n)$ for accessing elements by index (linear search)
Memory Allocation	Less frequent memory allocation/deallocation	More frequent memory allocation/deallocation

26. Explain the difference between the throw and throws keywords in Java.

Feature	throw	throws
Usage	Used to explicitly throw an exception within a method	Used in method signatures to declare exceptions that may be thrown by the method
Syntax	<code>`throw <exception>;`</code>	<code>`void methodName() throws <exception>;`</code>
Role	Used to raise an exception manually	Used to declare exceptions that a method might throw
Exception Type	Used with a specific exception instance	Used with exception class names
Example	<code>`throw new IllegalArgumentException();`</code>	<code>`void readFile() throws IOException;`</code>

java

```
// throw example
throw new IllegalArgumentException("Invalid input");

// throws example
void readFile() throws IOException {
    // Method implementation
}
```

27. Difference between == and .equals Method in java.

Feature	== Operator	.equals() Method
Usage	Compares references (memory addresses)	Compares the content or value of objects
Applicability	Used to compare primitive data types and objects	Used to compare objects
Implementation	Inherited from Object class	Overridden in classes to provide specific comparison
Purpose	Determines if two references point to the same object	Determines if two objects have the same content or value
Example	<code>`if (a == b)`</code>	<code>`if (a.equals(b))`</code>

java

```
String s1 = "hello";
String s2 = "hello";

// Using == operator (compares memory addresses)
System.out.println(s1 == s2); // true

// Using .equals() method (compares content)
System.out.println(s1.equals(s2)); // true
```

28. Explain Public Static Void Main

"public static void main" is a special line in Java programming. Here's what it means:

public : It means that this method (which is a block of code in Java) can be accessed from anywhere in the program.

static : This means the method belongs to the class itself, rather than any particular instance (object) of the class.

void : It means the method doesn't return any value after it is done running.

main : This is the name of the method. It's a special name because it's where the Java program starts running.

So, when you write `public static void main(String[] args)`, you're telling Java that this is the starting point of your program.

29. What is the meaning of import java.util.*

`import java.util.*;` essentially means that you want to use any classes or interfaces from the `java.util` package without specifying each one individually. This helps keep your code cleaner and more readable by grouping similar functionalities together from different packages.

Code Bashers