JAVA

- Instance variables are declared in a class, but outside a method, constructor or any block.
- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- **char[]**: char[] cannot be used directly in Java's collections (ArrayList, HashMap, etc.) because collections only work with objects.
- **Character[]**: Character[] can be used in collections like ArrayList<Character> because Character is a wrapper class and behaves like an object.
- **== cannot be overridden** as it is an operator.
- **equals()** can be overridden in user-defined classes to provide custom comparison logic.
- **equals()**: The default implementation (from the Object class) is identical to ==, but it can be overridden to compare object content.

JDK, JRE, and JVM:

JDK

JDK (Java Development Kit) is a Kit that provides the environment to **develop and execute(run)** the Java program. JDK is a kit(or package) that includes two things

- Development Tools(to provide an environment to develop your java programs)
- JRE (to execute your java program).

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

JRE

JRE (Java Runtime Environment) is an installation package that provides an environment to **only run(not develop)** the java program(or application)onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

JVM

JVM (Java Virtual Machine) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an ***interpreter*.**

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM Developer developed code > tested and executed in development machines. Execution phase require a medium for Java.

.class files are not native code (code which understands by cpu). To achieve WORA (Write Once Run Anywhere) concept can't be achieved if .java file is directly converted to native code.

Native code differs from OS to OS. So, Java created a intermediate file called .class and magical program called **"JVM"**. Its JVM's duty to convert .class into native code.

✹ What is Java?

Java is a high-level programming language that was developed by James Gosling in the year 1982. It is based on the principles of object-oriented programming and can be used to develop large-scale applications.

1. High-Level Languages

High-level programming languages are designed to be easy for humans to read, write, and understand. These languages provide a strong level of abstraction from the underlying machine hardware and focus on the logic of the application, abstracting away hardware details like memory management or CPU instructions.

High-level languages (like Java, Python, and JavaScript) provide **strong abstraction** from the hardware. These languages allow developers to focus on the logic and functionality of the application without needing to worry about how data is stored in memory or how specific CPU instructions are executed. Instead, the compiler or interpreter handles the communication with the machine's hardware.

Characteristics:

- **Easy to Read and Write**: Syntax is more human-readable, similar to English.
- **Portability**: Code written in high-level languages can run on different types of hardware with minimal changes.
- **Abstraction**: They hide the details of the computer's hardware, including memory management and CPU instructions.
- **Libraries and Frameworks**: They come with extensive libraries and frameworks to simplify common tasks like input/output operations, file handling, and more.
- **Examples**:

  - **C, C++**: Though closer to the hardware, these are still considered high-level.
  - **Java**: Write once, run anywhere (thanks to the JVM).
  - **Python**: Known for its simplicity and ease of use.
  - **JavaScript**: Used for web development and can interact with HTML/CSS.
  - **Ruby, Swift, Kotlin, PHP**: Other examples of high-level languages.

2. Low-Level Languages

Low-level programming languages provide little or no abstraction from the hardware. They are closely related to machine code, making them harder to read and write for humans but providing more control over the hardware.

Low-level languages (like Assembly or machine code) interact directly with the hardware, meaning the programmer needs to manage tasks like memory allocation, CPU instructions, and other hardware-specific details. This makes low-level languages powerful but difficult to work with, as you have to handle the intricacies of the hardware.

Types of Low-Level Languages:

- **Assembly Language**: Uses symbolic names (mnemonics) instead of binary code but still requires detailed knowledge of the hardware architecture.
- **Machin Language (Machine Code)**: The lowest-level language, consisting of binary (1s and 0s) instructions that the computer's CPU can execute directly.

Characteristics:

- **Harder to Read**: Written in symbolic or binary form that is difficult for humans to interpret.
- **Hardware-Specific**: Code is specific to the hardware for which it is written. Different architectures may require different assembly or machine code.
- **Efficient**: Offers more control over system resources like memory and CPU, making it suitable for writing system-level code, like operating systems, device drivers, and embedded systems.
- **Direct Interaction with Hardware**: Programmers have fine-grained control over CPU, memory, and registers.
- **Examples**:
  - **Assembly Language**: Uses mnemonics like MOV, ADD, SUB, etc.
  - **Machine Language**: Pure binary code executed by the CPU.
- **Key Differences**

Feature	High-Level Language	Low-Level Language
Abstraction	High abstraction from hardware	Little to no abstraction
Ease of use	Easy to write, read, and debug	Difficult to write and debug
Portability	Highly portable across platforms	Platform-specific (hardware-dependent)
Speed of Execution	Slower (due to abstraction)	Faster (closer to machine code)
Memory Management	Automatic (managed by the language)	Manual (requires explicit control)
Use Case	Application development, web, and mobile apps	System programming, device drivers, embedded systems
Examples	Python, Java, C++	Assembly, Machine Code

✹ Why is Java a platform independent language?

Java language was developed so that it does not depend on any hardware or software because the **compiler** compiles the code and then converts it to platform-independent byte code which can be run on multiple systems.

- The only condition to run that byte code is for the machine to have a runtime environment (JRE) installed in it.

Why is Java not a pure object-oriented language?

Java supports primitive data types - byte, boolean, char, short, int, float, long, and double and hence it is not a pure **object oriented language.**

Difference between Heap and Stack Memory in java. And how java utilizes this.

Stack memory is the portion of memory that was assigned to every individual program. And it was fixed. On the other hand, 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.

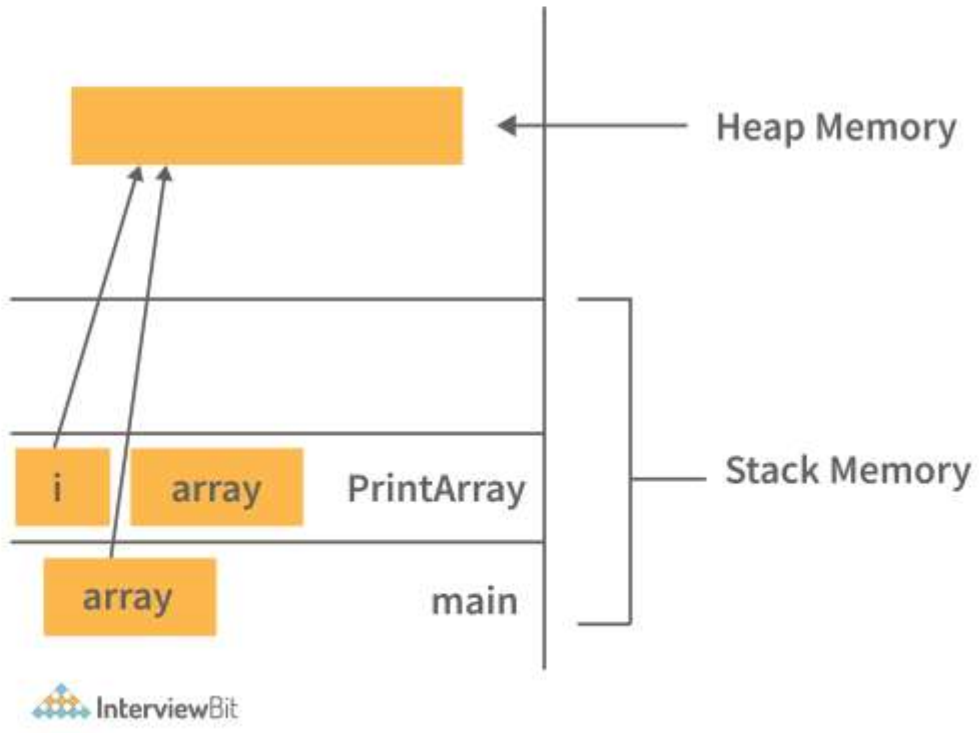
Java Utilizes this memory as -

- 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.

Example- **Consider the below java program**:

```
class Main {
  public void printArray(int[] array){
    for (int i: array)
      System.out.println(i);
  }
  public static void main(String args[]) {
    int[] array = new int[10];
    printArray(array);
  }
}
```

For this java program. The stack and heap memory occupied by java is -



Main() and PrintArray() is the method that will be available in the stack area and as well as the variables declared that will also be in the stack area.

And the Object (Integer Array of size 10) we have created, will be available in the Heap area because that space will be allocated to the program during runtime.

Pointers are used in C/ C++. Why does Java not make use of pointers?

Pointers are quite complicated and unsafe to use by beginner programmers. Java focuses on code simplicity, and the usage of pointers can make it challenging. Pointer utilization can also cause potential errors. Moreover, security is also compromised if pointers are used because the users can directly access memory with the help of pointers.

Thus, a certain level of abstraction is furnished by not including pointers in Java. Moreover, the usage of pointers can make the procedure of garbage collection quite slow and erroneous. Java makes use of references as these cannot be manipulated, unlike pointers.

What do you understand by an instance variable and a local variable?

Instance variables are those variables that are accessible by all the methods in the class. They are declared outside the methods and inside the class. These variables describe the properties of an object and remain bound to it at any cost.

All the objects of the class will have their copy of the variables for utilization. If any modification is done on these variables, then only that instance will be impacted by it, and all other class instances continue to remain unaffected.

Example:

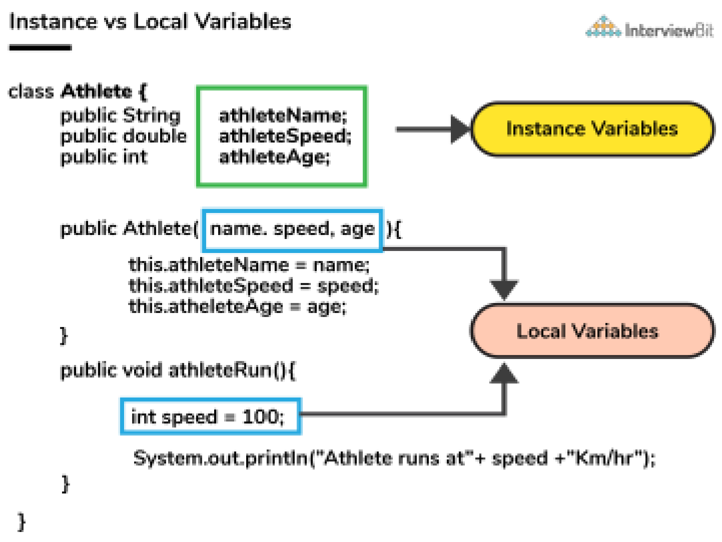
```
class Athlete {
public String athleteName;
public double athleteSpeed;
public int athleteAge;
}
```

Local variables are those variables present within a block, function, or constructor and can be accessed only inside them. The utilization of the variable is restricted to the block scope. Whenever a local variable is declared inside a method, the other class methods don't have any knowledge about the local variable.

Example:

```
public void athlete() {
```

String athleteName;
double athleteSpeed;
int athleteAge;
}



```
class Athlete {
        public String    athleteName;
        public double    athleteSpeed;
        public int       athleteAge;

        public Athlete( name. speed, age ){
                this.athleteName = name;
                this.athleteSpeed = speed;
                this.atheleteAge = age;
        }
        public void athleteRun(){

                int speed = 100;

                System.out.println("Athlete runs at"+ speed +"Km/hr");
        }
}
```

✸ In java primitive data type are not implemented as object, Explain?.

In Java, primitive data types are not implemented as objects, which means they are basic data types that are directly stored in memory and do not come with methods or advanced features that objects have. Here's what this distinction means:

Primitive Data Types vs. Objects:

1. **Primitive Data Types:**
    - Java has eight primitive data types: int, char, double, float, long, short, byte, and boolean.
    - These are not objects; they are basic, predefined types in Java, used to represent simple values.
    - They are stored directly in memory, and their values can be directly accessed without needing reference handling (i.e., no pointers or references).
    - Primitive types are more efficient in terms of performance because they are lightweight and do not have the overhead of object-related features (like method calls, object headers, etc.).
2. **Objects**:

- Objects are instances of classes, and they are more complex. Objects encapsulate data and behaviour (methods).
- When you work with objects, the object itself is stored in the **heap memory**, and variables store a **reference** to the object (not the object's value directly).
- Objects in Java are slower compared to primitive types due to their added complexity (like memory management, method invocation, etc.).

3. **No Methods Available**:

Unlike objects, primitive types do not have methods. For instance, you can call methods on objects (e.g., String.length()), but you cannot do the same on a primitive type. If you need methods for primitive-like values, Java provides **wrapper classes**.

What Does "Primitive Data Types Are Not Implemented as Objects" Mean?

1. **No Object Overhead:** Primitive types are not implemented as objects, meaning they do not come with the overhead associated with objects. They don't have built-in methods or properties, and they don't require memory management (like garbage collection) or referencing. This makes them faster and more memory-efficient for simple tasks like storing numerical values.
2. **Direct Value Storage:** When you declare a variable with a primitive type, the variable directly holds the value in the memory (stack memory), unlike objects where the variable holds a reference to the object in the heap.

✸ Wrapper Classes in Java:

To allow primitive types to behave like objects in some cases, Java provides wrapper classes for each primitive type. These wrapper classes are part of the java.lang package and are used when you need an object to represent a primitive value (e.g., for use in collections like ArrayList that work with objects only).

✸ **Primitive types and their corresponding wrapper classes:**

- int → Integer
- char → Character
- boolean → Boolean
- double → Double
- float → Float
- long → Long
- short → Short
- byte → Byte

✸ What is a constructor?

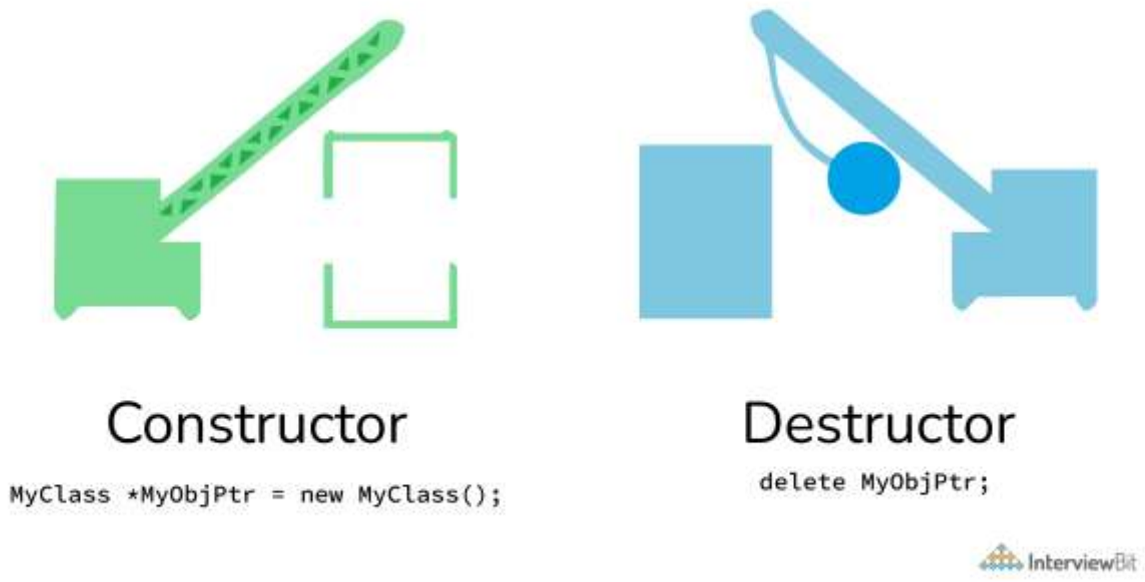
A **constructor** is a special function or method in object-oriented programming (OOP) that is automatically invoked when an instance (object) of a class is created. Its main purpose is to initialize the object's properties (or attributes) and set up any necessary setup logic for the object.

Key Points about Constructors:

1. **Initialization**: The primary job of a constructor is to initialize an object's data members when the object is created. It can take parameters to provide initial values to the object's attributes.
2. **Same Name as Class**: In many programming languages, such as C++, Java, and Python, the constructor has the same name as the class itself or follows a specific syntax for the constructor method (e.g., __init__ in Python).
3. **Automatic Invocation**: When you create an object from a class, the constructor is automatically called, and it doesn't need to be explicitly invoked.
4. **Overloading (in Some Languages)**: In languages like C++ and Java, constructors can be overloaded. This means you can have more than one constructor with different parameter lists, allowing objects to be initialized in multiple ways.
5. **No Return Type**: Constructors do not have a return type, not even void, because they are not called explicitly and do not return anything.

Constructors are special methods whose name is the same as the class name. The constructors serve the special purpose of initializing the objects.
For example, suppose there is a class with the name “MyClass”, then when you instantiate this class, you pass the syntax:
MyClass myClassObject = new MyClass();

Now here, the method called after “new” keyword - MyClass(), is the constructor of this class. This will help to instantiate the member data and methods and assign them to the object myClassObject.



```
MyClass *MyObjPtr = new MyClass();
```

```
delete MyObjPtr;
```

✹ What are the various types of constructors in C++?

The most common classification of constructors includes:

Default constructor: The default constructor is the constructor which doesn’t take any argument. It has no parameters.

```
class ABC
{
  int x;

  ABC()
  {
     x = 0;
  }
}
```

Parameterized constructor: The constructors that take some arguments are known as parameterized constructors.

```
class ABC
{
  int x;

  ABC(int y)
  {
     x = y;
  }
}
```

Copy constructor: A copy constructor is a member function that initializes an object using another object of the same class.

```
class ABC
```

```
{
  int x;

  ABC(int y)
  {
     x = y;
  }
  // Copy constructor
  ABC(ABC abc)
  {
     x = abc.x;
  }
}
```

✹ What is a copy constructor?

Copy Constructor is a type of constructor, whose purpose is to copy an object to another. What it means is that a copy constructor will clone an object and its values, into another object, is provided that both the objects are of the same class.

✹ What is a destructor?

Contrary to constructors, which initialize objects and specify space for them, Destructors are also special methods. But destructors free up the resources and memory occupied by an object. Destructors are automatically called when an object is being destroyed.

✹ Array in Java?

An **array** in Java is a data structure that allows you to store a fixed-size collection of elements of the same type. Arrays are useful when you want to group together multiple items of the same data type and access them using an index.

Key Characteristics of Arrays in Java:

1. **Fixed Size**: Once you declare the size of an array, it cannot be changed. The array has a fixed number of elements.
2. **Homogeneous**: All elements in an array must be of the same type (e.g., integers, strings, objects, etc.).
3. **Indexed**: Array elements are accessed using an index, starting from 0 for the first element and going up to n-1 (where n is the size of the array).

Declaring and Initializing Arrays:

Arrays can be declared and initialized in a few different ways.

1. **Declaration and Creation**:
   - Syntax:

```
dataType[] arrayName = new dataType[arraySize];
```

   - Example:

```
int[] numbers = new int[5];  // Creates an array of size 5 to store integers
```

2. **Declaration, Creation, and Initialization in One Step**:
   - Syntax:

```
dataType[] arrayName = { value1, value2, value3, ...};
```

Example:

```
int[] numbers = {1, 2, 3, 4, 5};  // Creates and initializes an array with 5 elements
```

3. **Accessing Elements**:
   - Array elements are accessed using the index, starting from 0.

Example:

```
int[] numbers = {10, 20, 30, 40};
System.out.println(numbers[0]);  // Output: 10
System.out.println(numbers[2]);  // Output: 30
```

4. **Modifying Elements**:
   - You can change the value of an element using its index.
   - Example:

```
int[] numbers = {1, 2, 3, 4};
numbers[1] = 10;  // Changes the second element (index 1) to 10
System.out.println(numbers[1]);  // Output: 10
```

Common Operations on Arrays:

1. **Iterating through an Array**:
   - Using a for loop:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int i = 0; i < numbers.length; i++) {
   System.out.println(numbers[i]);
}
```

   - Using an enhanced for loop (also called a "for-each" loop):

```
java
Copy code
for (int num : numbers) {
   System.out.println(num);
}
```

2. **Array Length**:
   - The number of elements in an array can be obtained using the length property.

Example:

```
int[] numbers = {1, 2, 3, 4, 5};
System.out.println(numbers.length);  // Output: 5
```

Types of Arrays:

1. **One-Dimensional Array**:
   - A single list of elements.

Example:

```
int[] arr = {10, 20, 30};
```

2. **Multi-Dimensional Array (e.g., 2D Array)**:
   - An array of arrays.

Example:

```
int[][] matrix = {
   {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println(matrix[1][2]);  // Output: 6
```

Advantages of Arrays:

- **Fast Access**: Array elements can be accessed directly using their index, making retrieval fast.
- **Fixed Size**: Memory allocation is efficient due to the fixed size.

Limitations of Arrays:

- **Fixed Size**: Once created, the size of an array cannot change. You need to know the size of the array in advance.
- **Homogeneous**: Arrays can only store elements of a single data type.

If dynamic resizing is needed, Java provides alternatives like ArrayList, which can grow or shrink as needed.

✹String in java?

In Java, a **String** is an object that represents a sequence of characters. Strings are widely used to store and manipulate text data. In Java, the String class is part of the java.lang package, and it is a built-in class that provides numerous methods for string manipulation.

Key Characteristics of Strings in Java:

1. **Immutable**: Strings in Java are immutable, meaning that once a String object is created, its value cannot be changed. Any operation that modifies a string will create a new String object.
2. **Stored in String Pool**: Java optimizes memory usage by storing string literals in a special memory area called the **String Pool**. If two strings have the same content, they will refer to the same memory location in the pool.

Creating Strings:

There are two primary ways to create strings in Java.

1. **Using String Literals**:
   - This is the most common way of creating strings. If the string already exists in the string pool, the reference to the existing string is returned.

   ```
   Example:
   String str1 = "Hello";
   String str2 = "Hello";  // str1 and str2 point to the same object in the string pool
   ```

2. **Using the new Keyword**:
   - This creates a new String object in the heap, even if the same value already exists in the string pool.

   ```
   Example:
   String str3 = new String("Hello");  // str3 is a new object in memory
   ```

String Methods:

The String class provides many methods for manipulating and inspecting string values. Some of the most commonly used methods include:

1. **Length of a String**:
    - The length() method returns the number of characters in the string.

Example:
String str = "Hello, World!";
System.out.println(str.length()); // Output: 13

2. **Concatenation**:
    - You can concatenate (join) two strings using the + operator or the concat() method.

Example:
String str1 = "Hello";
String str2 = "World";
String result = str1 + " " + str2; // Using + operator
System.out.println(result); // Output: Hello World

3. **Comparing Strings**:
    - equals(): Compares the content of two strings (case-sensitive).
    - equalsIgnoreCase(): Compares the content of two strings, ignoring case differences.
    - compareTo(): Compares two strings lexicographically (returns 0 if equal, a negative value if the first string is lexicographically smaller, and a positive value if larger).

Example:
String str1 = "Hello";
String str2 = "hello";
System.out.println(str1.equals(str2)); // Output: false
System.out.println(str1.equalsIgnoreCase(str2)); // Output: true

4. **Substring**:
    - The substring() method extracts a part of the string starting from a specific index.

Example:
String str = "Hello, World!";
String subStr = str.substring(7); // Extracts from index 7 to the end
System.out.println(subStr); // Output: World!

5. **Character Access**:
    - The charAt() method returns the character at a specific index.

Example:
String str = "Hello";
char ch = str.charAt(1); // Index 1 is 'e'
System.out.println(ch); // Output: e

6. **Replacing Characters**:
    - The replace() method replaces occurrences of a character or substring with another character or substring.

Example:

```
String str = "Hello, World!";
String newStr = str.replace('o', 'a');
System.out.println(newStr);  // Output: Hella, Warld!
```

7. **Splitting a String**:
   - The split() method splits a string into an array of substrings based on a given delimiter.

Example:

```
String str = "Java is fun";
String[] words = str.split(" ");
for (String word : words) {
   System.out.println(word);
}
// Output:
// Java
// is
// fun
```

8. **Trim**:
   - The trim() method removes leading and trailing whitespace from a string.

Example:

```
String str = "   Hello, World!   ";
System.out.println(str.trim());  // Output: Hello, World!
```

9. **Converting to Uppercase/Lowercase**:
   - toUpperCase(): Converts all characters to uppercase.
   - toLowerCase(): Converts all characters to lowercase.
   - Example:

```
String str = "Hello";
System.out.println(str.toUpperCase());  // Output: HELLO
System.out.println(str.toLowerCase());  // Output: hello
```

String Immutability:

As mentioned earlier, strings in Java are **immutable**. This means that any operation that appears to modify a string (e.g., concatenation, replacement) creates a new String object in memory rather than modifying the original string.

Example:

```
String str1 = "Hello";
String str2 = str1.concat(", World!");
System.out.println(str1);  // Output: Hello
System.out.println(str2);  // Output: Hello, World!
```

In the example above, str1 remains unchanged, and a new String object (str2) is created with the concatenated value.

StringBuilder and StringBuffer:

For cases where mutable strings are needed (e.g., when performing many modifications), Java provides two classes:

- **StringBuilder**: For creating and manipulating mutable strings (not thread-safe).
- **StringBuffer**: Similar to StringBuilder but is thread-safe (synchronized).

```
Example using StringBuilder:
StringBuilder sb = new StringBuilder("Hello");
sb.append(", World!");
System.out.println(sb.toString());  // Output: Hello, World!
```

Summary:

- **Strings in Java are immutable**, meaning they cannot be modified after creation.
- Strings can be created using literals or the new keyword.
- Java's String class provides many useful methods for string manipulation, including methods for concatenation, comparison, extraction of substrings, and modification.
- For performance-intensive string manipulation, use StringBuilder or StringBuffer.

✹ StringBuffer in java

In Java, **StringBuffer** is a class used to create mutable (modifiable) strings. Unlike String, which is immutable, a StringBuffer can be changed after it is created, making it more efficient for certain operations like concatenation or modifying strings multiple times.

Here's a breakdown of StringBuffer in Java:

Key Characteristics of StringBuffer:

1. **Mutable:** You can modify the contents of a StringBuffer object without creating a new object.
2. **Thread-safe:** StringBuffer methods are synchronized, meaning they are safe to use in multithreaded environments. However, this makes it slightly slower than StringBuilder (another mutable string class).
3. **Dynamic resizing:** When concatenating strings or adding characters, the StringBuffer automatically grows in size as needed.
4. **Efficient for string manipulations:** Operations like appending, inserting, or deleting characters in a StringBuffer are more efficient than using a String.

Commonly Used Methods in StringBuffer:

1. **append(String str)**: Adds the specified string to the end of the current string.

```
StringBuffer sb = new StringBuffer("Hello");
```

```
sb.append(" World");
System.out.println(sb);  // Output: Hello World
```

2. **insert(int offset, String str)**: Inserts a string at the specified position.

```
StringBuffer sb = new StringBuffer("Hello");
sb.insert(5, " Java");
System.out.println(sb);  // Output: Hello Java
```

3. **delete(int start, int end)**: Removes characters between the start and end positions.

```
StringBuffer sb = new StringBuffer("Hello World");
sb.delete(5, 11);
System.out.println(sb);  // Output: Hello
```

4. **replace(int start, int end, String str)**: Replaces the characters between the start and end positions with the specified string.

```
StringBuffer sb = new StringBuffer("Hello World");
sb.replace(6, 11, "Java");
System.out.println(sb);  // Output: Hello Java
```

5. **reverse()**: Reverses the order of the characters in the StringBuffer.

```
StringBuffer sb = new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);  // Output: olleH
```

6. **capacity()**: Returns the current capacity (the amount of memory reserved for the StringBuffer).

```
StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity());  // Default: 16
```

7. **setLength(int newLength)**: Sets the length of the StringBuffer. If the new length is greater, null characters (\u0000) are added; if it's smaller, the string is truncated.

```
StringBuffer sb = new StringBuffer("Hello World");
sb.setLength(5);
System.out.println(sb);  // Output: Hello

```

8. **toString()**: Converts the StringBuffer to a regular String.

```
StringBuffer sb = new StringBuffer("Hello World");
String result = sb.toString();
System.out.println(result);  // Output: Hello World
```

Example Usage:

```
public class Main {
    public static void main(String[] args) {
        // Creating a StringBuffer object
        StringBuffer sb = new StringBuffer("Welcome");

        // Appending text to the StringBuffer
        sb.append(" to Java");
        System.out.println(sb);  // Output: Welcome to Java

        // Inserting text at a specific position
        sb.insert(7, " everyone,");
        System.out.println(sb);  // Output: Welcome everyone, to Java

        // Reversing the StringBuffer
        sb.reverse();
        System.out.println(sb);  // Output: avaJ ot ,enoyreve emocleW
    }
}
```

Difference Between String, StringBuffer, and StringBuilder:

1. **String:**
    - Immutable (cannot be changed once created).
    - Slow for multiple modifications (because it creates new objects).
2. **StringBuffer:**
    - Mutable (can be changed).
    - Thread-safe (synchronized).
    - Slightly slower than StringBuilder due to synchronization overhead.
3. **StringBuilder:**
    - Mutable (like StringBuffer).
    - Not thread-safe (no synchronization).
    - Faster than StringBuffer for single-threaded applications.

When to Use StringBuffer:

- Use StringBuffer when you need a mutable string and are working in a **multithreaded** environment where multiple threads might need to access and modify the same string.
- If you don’t need synchronization (i.e., thread safety), it's better to use StringBuilder for faster performance.

Summary:

StringBuffer is a useful class in Java for mutable strings, especially when you need thread safety and need to modify strings repeatedly, such as appending, deleting, or inserting characters.

✸String builder in Java

In Java, **StringBuilder** is a class used to create mutable (modifiable) sequences of characters. Unlike the String class, which creates immutable strings, StringBuilder allows you to modify the contents of the string without creating new objects. This makes StringBuilder more efficient when you need to perform many modifications (such as appending, inserting, or replacing characters) on a string.

Key Characteristics of StringBuilder:

1. **Mutable**: You can change the contents of a StringBuilder object after it has been created, such as by appending, inserting, or deleting characters.
2. **Efficient for String Modifications**: Since StringBuilder does not create new objects every time it is modified (unlike String), it is more memory-efficient for scenarios where strings undergo frequent changes.
3. **Not Thread-Safe**: StringBuilder is not synchronized, meaning it is not thread-safe. If multiple threads are accessing the same StringBuilder object, there could be concurrency issues. If thread safety is required, use StringBuffer instead.

Creating a StringBuilder:

You can create a StringBuilder object in different ways:

1. **Empty StringBuilder**:
   - Creates a StringBuilder with an initial capacity of 16 characters.

Example:
StringBuilder sb = new StringBuilder();

2. **StringBuilder with Initial Content**:
   - You can create a StringBuilder with a string, and it will have an initial capacity large enough to hold the string and additional characters.

Example:
StringBuilder sb = new StringBuilder("Hello");

Common StringBuilder Methods:

1. **append()**:
   - Adds (appends) the specified string or character to the end of the current StringBuilder.

Example:
StringBuilder sb = new StringBuilder("Hello");
sb.append(", World!");
System.out.println(sb.toString()); // Output: Hello, World!

2. **insert()**:
   - Inserts a string or character at the specified index.

Example:
StringBuilder sb = new StringBuilder("Hello!");
sb.insert(5, ", World"); // Inserts at index 5
System.out.println(sb.toString()); // Output: Hello, World!

3. **delete()**:
   - Removes a sequence of characters from the StringBuilder, between the start and end indices (end index is exclusive).

Example:
StringBuilder sb = new StringBuilder("Hello, World!");
sb.delete(5, 7); // Deletes the comma and space
System.out.println(sb.toString()); // Output: HelloWorld!

4. **replace()**:
   - Replaces characters in the StringBuilder between two indices with a new string.

Example:
StringBuilder sb = new StringBuilder("Hello, World!");
sb.replace(7, 12, "Java"); // Replaces "World" with "Java"
System.out.println(sb.toString()); // Output: Hello, Java!

5. **reverse()**:
   - Reverses the entire sequence of characters in the StringBuilder.

Example:
StringBuilder sb = new StringBuilder("Hello");
sb.reverse();
System.out.println(sb.toString()); // Output: olleH

6. **charAt()**:
   - Returns the character at a specific index.

Example:
StringBuilder sb = new StringBuilder("Hello");
char ch = sb.charAt(1); // Retrieves character at index 1
System.out.println(ch); // Output: e

7. **setCharAt()**:
   - Changes the character at the specified index.

Example:
StringBuilder sb = new StringBuilder("Hello");
sb.setCharAt(1, 'a'); // Changes 'e' to 'a'
System.out.println(sb.toString()); // Output: Hallo

8. **length() and capacity()**:
   - length(): Returns the number of characters in the StringBuilder.

- capacity(): Returns the current capacity (the amount of storage available for the characters).

Example:

```
StringBuilder sb = new StringBuilder("Hello");
System.out.println(sb.length());   // Output: 5
System.out.println(sb.capacity()); // Output: 21 (initial capacity plus the length of "Hello")
```

9. **toString()**:
   - Converts the StringBuilder content back into a String.

Example:

```
StringBuilder sb = new StringBuilder("Hello");
String str = sb.toString();  // Converts to a String
System.out.println(str);  // Output: Hello
```

Example Usage:

```
public class Main {
    public static void main(String[] args) {
        // Creating a StringBuilder
        StringBuilder sb = new StringBuilder("Hello");

        // Append a string
        sb.append(" World");

        // Insert a string
        sb.insert(5, ",");

        // Replace part of the string
        sb.replace(6, 11, "Java");

        // Reverse the string
        sb.reverse();

        // Print the final string
        System.out.println(sb.toString());  // Output: avaJ ,olleH
    }
}
```

Advantages of StringBuilder:

- **Efficiency**: StringBuilder is more efficient than String when performing multiple modifications like appending or replacing characters, as it doesn't create new objects for every change.
- **Mutability**: You can modify the content without creating new objects.
- **Performance**: It improves performance, especially in scenarios where strings are frequently modified (e.g., in loops).

When to Use StringBuilder:

- **Use StringBuilder** when you need to perform many string modifications, such as appending, inserting, or replacing characters in a string.
- **Use String** when you are working with constant or unchanging text, as String is simpler and less resource-intensive for basic usage.

In summary, StringBuilder is a powerful tool for handling mutable strings efficiently in Java, especially in situations where performance is critical.

✹How to convert String to char Array

In Java, you can take input for both String and char[] (character array) using the Scanner class or BufferedReader. Here's how to do it:

Using Scanner

1. **String input:** You can use nextLine() to take a string input.
2. **char[] input:** You can read a string first and then convert it to a character array using toCharArray().

Here's an example:

```
import java.util.Scanner;

public class InputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Taking String input
        System.out.print("Enter a string: ");
        String strInput = scanner.nextLine();
        System.out.println("String input: " + strInput);

        // Taking char[] input
        System.out.print("Enter a string to convert to char array: ");
        String charArrayInput = scanner.nextLine();
        char[] charArray = charArrayInput.toCharArray(); // Convert to char array

        System.out.println("Char array input: ");
        for (char c : charArray) {
            System.out.print(c + " ");
        }
        scanner.close();
    }
}
```

Using BufferedReader

BufferedReader can also be used for larger inputs or for better performance:

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class InputExampleBufferedReader {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        // Taking String input
        System.out.print("Enter a string: ");
        String strInput = reader.readLine();
        System.out.println("String input: " + strInput);

        // Taking char[] input
        System.out.print("Enter a string to convert to char array: ");
        String charArrayInput = reader.readLine();
        char[] charArray = charArrayInput.toCharArray(); // Convert to char array

        System.out.println("Char array input: ");
        for (char c : charArray) {
            System.out.print(c + " ");
        }
    }
}
```

Explanation:

- scanner.nextLine(): Takes the whole line as a string.
- string.toCharArray(): Converts the string into a character array (char[]).

These approaches allow you to handle both string and character array inputs.

✹How to convert char array to string

To convert a char[] (character array) to a String in Java, you can use one of the following approaches:

1. Using the String Constructor

The simplest way is to pass the char[] to the String constructor.

```java
char[] charArray = {'H', 'e', 'l', 'l', 'o'};
String str = new String(charArray);
System.out.println(str); // Output: Hello
```

2. Using String.valueOf() Method

This is another common way to convert a char[] to a String.

```java
char[] charArray = {'W', 'o', 'r', 'l', 'd'};
String str = String.valueOf(charArray);
System.out.println(str); // Output: World
```

3. Using StringBuilder or StringBuffer

If you are manipulating characters before converting them into a String, you can append them to a StringBuilder (or StringBuffer) and then call the toString() method.

```
char[] charArray = {'J', 'a', 'v', 'a'};
StringBuilder sb = new StringBuilder();
for (char c : charArray) {
    sb.append(c);
}
String str = sb.toString();
System.out.println(str); // Output: Java
```

Summary:

- **Constructor**: new String(charArray) is direct and simple.
- **String.valueOf(charArray)**: A static method that returns a String representation.
- **StringBuilder**: Useful if you want to modify or build a String dynamically.

✹Metods to split string

In Java, there are several ways to split a string into an array of substrings. The most commonly used methods are split(), StringTokenizer, and Pattern.split(). Here are different ways to split a string:

1. Using String.split() Method

The split() method splits a string based on a specified regular expression (regex) and returns an array of substrings.

Example: Split by space

```
public class SplitExample {
    public static void main(String[] args) {
        String str = "Hello World from Java";

        // Split by space
        String[] words = str.split(" ");

        for (String word : words) {
            System.out.println(word);
        }
    }
}
```

Output:

```
Hello
World
from
Java
```

Example: Split by a comma

```
public class SplitByComma {
    public static void main(String[] args) {
        String str = "apple,banana,orange";

        // Split by comma
        String[] fruits = str.split(",");

        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

Output:

```

apple
banana
orange
```

2. Using StringTokenizer Class

The StringTokenizer class allows you to split a string into tokens based on a specified delimiter. It is a legacy class but still works well for simple string splitting.

Example: Using StringTokenizer

```
import java.util.StringTokenizer;

public class TokenizerExample {
    public static void main(String[] args) {
        String str = "Hello,World,from,Java";

        // Create a StringTokenizer with a comma as the delimiter
        StringTokenizer tokenizer = new StringTokenizer(str, ",");

        while (tokenizer.hasMoreTokens()) {
            System.out.println(tokenizer.nextToken());
        }
    }
}
```

Output:

```
Hello
World
from
Java
```

3. Using Pattern.split() Method

The Pattern.split() method allows you to split a string using more complex regular expressions. This is part of the java.util.regex package.

Example: Using Pattern.split()

```
import java.util.regex.Pattern;

public class PatternSplitExample {
    public static void main(String[] args) {
        String str = "apple,banana;orange:grape";

        // Use multiple delimiters (comma, semicolon, colon)
        Pattern pattern = Pattern.compile("[,;:]");
        String[] fruits = pattern.split(str);

        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```
Output:

```
Copy code
apple
banana
orange
grape
```

4. Using substring() Method in Combination with indexOf()

While substring() is not technically a splitting method, you can use it along with indexOf() to manually split a string.

Example: Using substring() and indexOf()

```
public class SubstringSplitExample {
    public static void main(String[] args) {
        String str = "apple,banana,orange";

        while (str.contains(",")) {
            int commaIndex = str.indexOf(",");
            String word = str.substring(0, commaIndex);
            System.out.println(word);
            str = str.substring(commaIndex + 1);
        }
        System.out.println(str); // Print the last word
    }
}
```
Output:

```
Copy code
apple
banana
orange
```

5. Using Scanner Class

The Scanner class can also be used to tokenize or split a string based on a delimiter.

Example: Using Scanner

```
import java.util.Scanner;

public class ScannerSplitExample {
    public static void main(String[] args) {
        String str = "apple orange banana grape";

        // Create a Scanner with space as the delimiter
        Scanner scanner = new Scanner(str);

        while (scanner.hasNext()) {
            System.out.println(scanner.next());
        }

        scanner.close();
    }
}
```

Output:

```
apple
orange
banana
grape
```

Summary of Methods:

1. **split()**: Most common method, allows splitting with regular expressions.
2. **StringTokenizer**: Simple tokenization using delimiters.
3. **Pattern.split()**: More flexible splitting with complex regex patterns.
4. **substring() and indexOf()**: Manual splitting by searching for delimiters.
5. **Scanner**: Tokenizes input based on a delimiter.

Choose the method based on your specific use case and requirements.

✹Methods to compare two strings

In Java, there are multiple ways to compare two strings. The comparison can be based on content, lexicographical order, or case sensitivity. Below are the most commonly used methods for comparing two strings:

1. Using equals() Method

The equals() method compares the actual content of two strings. It returns true if the strings are exactly the same (case-sensitive).

Example:

```
public class StringComparison {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "Hello";
        String str3 = "hello";

        // Compare strings using equals() (case-sensitive)
        System.out.println(str1.equals(str2)); // true
        System.out.println(str1.equals(str3)); // false
    }
}
```

- **equals()** performs a case-sensitive comparison.

---

2. Using equalsIgnoreCase() Method

The equalsIgnoreCase() method compares two strings while ignoring case differences.

Example:

```
public class StringComparisonIgnoreCase {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "hello";

        // Compare strings using equalsIgnoreCase() (case-insensitive)
        System.out.println(str1.equalsIgnoreCase(str2)); // true
    }
}
```

- **equalsIgnoreCase()** compares the strings without considering case, so "Hello" and "hello" are treated as equal.

---

3. Using compareTo() Method

The compareTo() method compares two strings lexicographically. It returns:

- 0 if the strings are equal.
- A negative value if the first string is lexicographically less than the second.
- A positive value if the first string is lexicographically greater than the second.

Example:

```
public class StringCompareTo {
    public static void main(String[] args) {
        String str1 = "Apple";
        String str2 = "Banana";
        String str3 = "Apple";

        // Compare strings using compareTo() (lexicographical comparison)
        System.out.println(str1.compareTo(str2)); // Negative (Apple < Banana)
        System.out.println(str1.compareTo(str3)); // 0 (Apple == Apple)
    }
}
```

- **compareTo()** performs lexicographical comparison, considering the natural order of characters based on their Unicode values.

---

4. Using compareToIgnoreCase() Method

The compareToIgnoreCase() method is similar to compareTo(), but it ignores case when comparing strings lexicographically.

Example:

```
public class StringCompareToIgnoreCase {
    public static void main(String[] args) {
        String str1 = "Apple";
        String str2 = "apple";

        // Compare strings using compareToIgnoreCase() (case-insensitive lexicographical comparison)
        System.out.println(str1.compareToIgnoreCase(str2)); // 0 (equal when case is ignored)
    }
}
```

- **compareToIgnoreCase()** performs lexicographical comparison ignoring case.

---

5. Using contentEquals() Method

The contentEquals() method compares a string with a StringBuffer or StringBuilder content, checking if the content matches exactly.

Example:

```
public class StringContentEquals {
    public static void main(String[] args) {
        String str1 = "Hello";
        StringBuilder sb = new StringBuilder("Hello");

        // Compare string with StringBuilder content using contentEquals()
        System.out.println(str1.contentEquals(sb)); // true
    }
}
```

- **contentEquals()** checks for exact matches, including case sensitivity, between a string and a StringBuffer or StringBuilder.

---

6. Using regionMatches() Method

The regionMatches() method compares a specific region (substring) of two strings. There are two versions:

- **regionMatches(int toffset, String other, int ooffset, int len)**: Case-sensitive comparison of regions.
- **regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)**: Optionally case-insensitive comparison of regions.

Example:

```
public class StringRegionMatches {
    public static void main(String[] args) {
        String str1 = "HelloWorld";
        String str2 = "world";

        // Compare a substring region of str1 with str2 (case-sensitive)
        System.out.println(str1.regionMatches(5, str2, 0, 5)); // false

        // Compare regions ignoring case
        System.out.println(str1.regionMatches(true, 5, str2, 0, 5)); // true
    }
}
```

- **regionMatches()** is useful for comparing specific parts of two strings.

---

7. Using Objects.equals() Method

If you want to avoid null pointer exceptions when comparing strings, you can use Objects.equals(). It safely compares two strings, even if one or both are null.

Example:

```
import java.util.Objects;

public class StringObjectsEquals {
    public static void main(String[] args) {
        String str1 = null;
        String str2 = "Hello";

        // Compare using Objects.equals() (handles null)
        System.out.println(Objects.equals(str1, str2)); // false
        System.out.println(Objects.equals(str1, null)); // true
    }
}
```

- **Objects.equals()** safely handles null comparisons.

---

8. Using matches() Method (for Regex)

If you need to compare strings using a regular expression, the matches() method is useful. It returns true if the string matches the given regular expression.

Example:

```
public class StringMatches {
    public static void main(String[] args) {
        String str = "Hello123";

        // Check if the string matches a regex pattern
        System.out.println(str.matches("[A-Za-z]+[0-9]+")); // true (letters followed by numbers)
    }
}
```

- **matches()** uses regex to compare strings.

Summary:

- **equals()**: Case-sensitive content comparison.
- **equalsIgnoreCase()**: Case-insensitive content comparison.
- **compareTo()**: Lexicographical comparison.
- **compareToIgnoreCase()**: Case-insensitive lexicographical comparison.
- **contentEquals()**: Compares with StringBuffer or StringBuilder.
- **regionMatches()**: Compares a substring region.
- **Objects.equals()**: Safe comparison, handling null.
- **matches()**: Compares with regular expression.

Each method serves a specific purpose depending on your needs for case-sensitivity, lexicographical ordering, or even null safety.

✸Is toString() compulsory to use in StringBuilder

No, using toString() is not **compulsory** when working with StringBuilder, but it is often necessary when you need to convert the StringBuilder object to a String.

Why Use toString() with StringBuilder?

- **StringBuilder stores character sequences**, and its toString() method converts this sequence into an immutable String. If you want to interact with a method or API that expects a String (since many methods in Java work with String objects), you need to use toString() to convert the StringBuilder content into a String.

When You Don't Need toString():

If you are just manipulating or appending strings using the StringBuilder, you can continue working with it without converting to a String. The StringBuilder methods (like append(), insert(), reverse()) operate directly on the mutable character sequence within the StringBuilder, so toString() is not required for string manipulation.

When You Need toString():

If you need to return the content of the StringBuilder or pass it to a method that accepts a String, then you need to call toString().

Example:

Without toString() (manipulating the string):

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");  // Modifying the StringBuilder
System.out.println(sb);  // This will automatically print the current content of sb: "Hello World"
```

With toString() (when converting to String):

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");

// If you need to return or pass it as a String, you need to use toString()
String result = sb.toString();
System.out.println(result);  // Output: "Hello World"
```

Conclusion:

- **Not compulsory** for internal manipulation using StringBuilder.
- **Necessary** when you need to convert StringBuilder to a String, for example, when passing it to a method that expects a String.

✸Difference between character[] and char[]

In Java, char[] and Character[] represent arrays of characters, but they have some key differences due to the distinction between primitive types (char) and wrapper classes (Character). Here's a breakdown of the differences:

1. Type: Primitive vs. Object

- **char[]**:
    - This is an array of primitive type char.
    - char is a basic data type that holds a single 16-bit Unicode character.
    - Elements in a char[] array are primitive values, not objects.

- **Character[]**:
    - This is an array of the wrapper class Character.
    - Character is an object that wraps the primitive char type.
    - Each element in a Character[] array is an object, which provides additional methods and functionality beyond what a primitive char offers.

2. Memory Usage

- **char[]**:
  - Requires less memory because it stores primitive char values directly.
  - A primitive char takes 2 bytes of memory.
- **Character[]**:
  - Requires more memory because it stores Character objects.
  - Each Character object wraps a char and adds additional overhead, such as the object header.

3. Autoboxing and Unboxing

- **char[]**:
  - No autoboxing/unboxing occurs since it's a primitive type.
- **Character[]**:
  - Autoboxing is the process where Java automatically converts a char primitive to a Character object.
  - Unboxing is when a Character object is converted back to a char primitive.
  - This means Character[] can have performance overhead due to autoboxing/unboxing when interacting with primitive char.

4. Nullability

- **char[]**:
  - Elements of char[] cannot be null. The default value for uninitialized elements is '\u0000' (the null character).
- **Character[]**:
  - Elements in a Character[] array can be null, because Character is a reference type (an object).

5. Methods and Utility

- **char[]**:
  - As a primitive type, char doesn't have methods. You have to manipulate the char[] directly using array syntax and loops.
- **Character[]**:
  - Since Character is an object, it has methods, such as isDigit(), isLetter(), etc., that you can call on individual elements of a Character[].

6. Use in Collections

- **char[]**:
  - char[] cannot be used directly in Java's collections (ArrayList, HashMap, etc.) because collections only work with objects.
- **Character[]**:
  - Character[] can be used in collections like ArrayList<Character> because Character is a wrapper class and behaves like an object.

7. Example Code:

```
e
    public static void main(String[] args) {
        // char[] Example
        char[] charArray = {'a', 'b', 'c'};
        for (char c : charArray) {
            System.out.println(c);  // Output: a b c
        }

        // Character[] Example
        Character[] characterArray = {'x', 'y', 'z'};
        for (Character c : characterArray) {
            System.out.println(c);  // Output: x y z
        }

        // Autoboxing example
        Character[] chars = new Character[2];
        chars[0] = 'A';  // char 'A' is autoboxed to Character
        System.out.println(chars[0]);
    }
}
```

Summary:

- **char[]** is an array of primitive char values, more memory efficient but with limited functionality.
- **Character[]** is an array of Character objects, allowing for null values and additional methods but with greater memory overhead and performance costs due to autoboxing/unboxing.

✹ Static keyword

The static keyword in Java is used for **class-level** features. When a member (variable, method, block, or nested class) is declared static, it belongs to the class itself rather than to instances (objects) of the class. This means static members are shared across all instances of the class and can be accessed without creating an instance.

Use of static Keyword in Java:

1. **Static Variables (Class Variables)**:
    - A static variable is shared among all instances of a class. There is only **one copy** of a static variable, regardless of how many objects are created.
    - Static variables are usually used for constants or to share a value across all instances of a class.

Example:

```
public class Example {
  // Static variable
  static int count = 0;
  public Example() {
    count++;
  }
  public static void main(String[] args) {
    Example obj1 = new Example();
    Example obj2 = new Example();
    System.out.println("Number of objects created: " + Example.count); // Output: 2
  }
}
```

- In this example, the count variable is shared by all objects and keeps track of how many objects of Example have been created.

2. **Static Methods**:

   - A static method belongs to the class rather than any instance of the class. It can be called **without creating an object** of the class.
   - Static methods can only access static data (static variables or other static methods). They **cannot access instance variables** or methods directly since instance variables are tied to individual objects.

Example:

```
public class Example {
  static int count = 0;
  // Static method
  public static void displayCount() {
    System.out.println("Count: " + count);
  }
  public static void main(String[] args) {
    Example.displayCount();  // Call static method without creating an instance
  }
}
```

   - In this example, the static method displayCount() is called directly using the class name Example, without needing to create a n instance.

3. **Static Block (Static Initialization Block)**:

   - A static block is used for **static initialization** of a class. It is executed when the class is loaded into memory, before the main method or any static method is invoked.
   - It is mainly used to initialize static variables that require more complex logic than a simple assignment.

Example:

```
public class Example {
  static int value;
  // Static block
  static {
    value = 10;
    System.out.println("Static block executed. Value is initialized to " + value);
  }
  public static void main(String[] args) {
    System.out.println("Main method executed.");
    System.out.println("Value: " + value);
  }
}
```

- The static block in this example initializes the static variable value when the class is loaded, before the main method is executed.

4. **Static Classes (Nested Static Classes)**:
   - A class can be declared static, but only if it is a **nested class** (a class within another class). A static nested class **does not** have a reference to the instance of the enclosing class.
   - This is useful for creating helper classes or grouping related classes together.

Static Nested Classes in Java

In Java, a class can be nested within another class, and if it is declared **static**, it is known as a **static nested class**. Unlike inner classes (non-static nested classes), a **static nested class** does not have a reference to an instance of its enclosing (outer) class. Here's a detailed explanation of static nested classes:

Key Characteristics:

1. **Declaration**: A static nested class is declared inside another class but is prefixed with the static keyword. This means that the static nested class behaves similarly to other static members of the enclosing class (like static methods or fields).

```
public class OuterClass {
    static class NestedStaticClass {
        // Static nested class content
    }
}
```

2. **No Reference to Outer Class**:
   - A static nested class **cannot** access instance members (non-static members) of the outer class directly. It can only access **static members** of the enclosing class because it is not tied to an instance of the outer class.
   - It does not hold a reference to an instance of the outer class, meaning it can exist and be used independently of an object of the outer class.

3. **Instantiation**: To instantiate a static nested class, you **do not need** an instance of the outer class. You can create an instance of the static nested class directly by referencing the outer class name.

```
public class OuterClass {
    static class NestedStaticClass {
        void display() {
            System.out.println("Inside static nested class");
        }
    }
}

// Creating an instance of static nested class
public class Main {
    public static void main(String[] args) {
        OuterClass.NestedStaticClass nested = new OuterClass.NestedStaticClass();
        nested.display();
    }
}
```

4. **Access to Outer Class Static Members**: A static nested class can access the static fields and methods of the outer class because static members belong to the class, not to any instance.

```java
public class OuterClass {
    private static String outerStaticVar = "Outer static variable";

    static class NestedStaticClass {
        void display() {
            System.out.println("Accessing: " + outerStaticVar);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass.NestedStaticClass nested = new OuterClass.NestedStaticClass();
        nested.display(); // Output: Accessing: Outer static variable
    }
}
```

5. **Use Cases**:
    - **Helper Classes**: Static nested classes are often used to group **helper or utility classes** that are closely related to the outer class.
    - **Encapsulation**: It can be used to logically group classes that should only be used in the context of the outer class while keeping them separated to maintain a clean code structure.
    - **Independent Functionality**: Since static nested classes are not dependent on an instance of the outer class, they are used when the nested class provides functionality that does not rely on the outer class's instance variables or methods.
6. **Example: Static Nested Class as a Helper Class**:

```java
public class Calculator {

    // Static nested class to define math operations
    static class Operations {
        static int add(int a, int b) {
            return a + b;
        }

        static int subtract(int a, int b) {
            return a - b;
        }
    }

    public static void main(String[] args) {
        // Use the static nested class without needing an instance of Calculator
        int result = Calculator.Operations.add(5, 3);
        System.out.println("Result of addition: " + result);  // Output: 8
    }
}
```

Key Points:

- **Static nested classes** are useful when the nested class logically belongs to the outer class but doesn't need access to the outer class's instance variables or methods.
- **Instantiation**: A static nested class is instantiated without an instance of the enclosing class.

- **Encapsulation**: Static nested classes help with encapsulation, keeping classes grouped and organized within a meaningful context.

  By using static nested classes, you can organize your code better and avoid clutter, especially when you want to group related classes together.

Example:

```
public class OuterClass {
    static int outerStaticVar = 10;
    // Static nested class
    static class NestedClass {
        void display() {
            System.out.println("Static variable from outer class: " + outerStaticVar);
        }
    }
    public static void main(String[] args) {
        OuterClass.NestedClass nestedObj = new OuterClass.NestedClass();
        nestedObj.display();
    }
}
```

  - In this example, NestedClass is a static nested class and can access the static members of the outer class, but it does not need an instance of OuterClass to be created.

Key Points about static in Java:

- **Static Variables**:
  - Shared by all instances of a class.
  - There is only one copy of a static variable, no matter how many objects of the class are created.
- **Static Methods**:
  - Can be called without creating an object of the class.
  - Can only access static data directly (static variables, static methods).
  - Cannot access non-static (instance) variables or methods.
- **Static Block**:
  - Used for static initialization.
  - Executes when the class is loaded into memory, before any methods are called.
- **Static Nested Class**:
  - Can be used to logically group classes that are only used in one place.
  - A static nested class does not have access to non-static members of the outer class.

Example Summarizing All Uses of static:

```
public class Example {
    // Static variable
    static int staticVar = 100;

    // Static block
    static {
        System.out.println("Static block executed");
        staticVar = 50;
    }

    // Static method
    public static void staticMethod() {
        System.out.println("Static method called, staticVar = " + staticVar);
    }

    // Static nested class
    static class StaticNestedClass {
        void display() {
            System.out.println("Inside static nested class. StaticVar: " + staticVar);
        }
    }

    public static void main(String[] args) {
        // Access static variable
        System.out.println("Static variable: " + staticVar);

        // Call static method
        staticMethod();

        // Use static nested class
        StaticNestedClass nestedObject = new StaticNestedClass();
        nestedObject.display();
    }
}
```

Output:

```
Static block executed
Static variable: 50
Static method called, staticVar = 50
Inside static nested class. StaticVar: 50
```

Summary:

- **static variables**: Shared among all instances, belong to the class rather than any object.
- **static methods**: Can be called without creating an instance, but can only access static data.
- **static block**: Executes once when the class is loaded, used for initializing static data.
- **static nested class**: A nested class that can access static members of the outer class but not instance members directly.

✹Use cases of static keyword

The static keyword is used in various programming languages like Java, C++, C#, and JavaScript to define members (variables, methods, or classes) that belong to a class or context rather than to individual instances of the class. Below are some common **use cases** for the static keyword:

1. Shared Class-Level Variables

- **Use Case**: When you want a variable to be shared across all instances of a class rather than each instance having its own copy.

- **Example (Java)**:

```
class Counter {
  static int count = 0; // Shared across all instances
  Counter() {
    count++;
  }
}
Counter c1 = new Counter();
Counter c2 = new Counter();
System.out.println(Counter.count); // Output: 2
```

2. Static Methods

- **Use Case**: Methods that can be called without creating an instance of the class. These methods usually operate on static variables or perform general utility tasks.
- **Example (Java)**:

```
class MathUtil {
  static int square(int x) {
    return x * x;
  }
}
System.out.println(MathUtil.square(5)); // Output: 25
```

3. Utility or Helper Classes

- **Use Case**: Defining a class with all static methods to perform utility functions (e.g., Math class in Java or utility libraries like Lodash in JavaScript).
- **Example (JavaScript)**:

```
Javascript-Code
class Utils {
  static add(a, b) {
    return a + b;
  }
}
console.log(Utils.add(2, 3)); // Output: 5
```

4. Static Blocks for Initialization

- **Use Case**: In languages like Java and C#, static blocks can be used to initialize static variables or execute some code once, when the class is first loaded.
- **Example (Java)**:

```
class DatabaseConfig {
  static String url;
  static {
    url = "jdbc:mysql://localhost:3306/mydb"; // Static block initialization
  }
}
System.out.println(DatabaseConfig.url); // Output: "jdbc:mysql://localhost:3306/mydb"
```

5. Accessing Class Constants

- **Use Case**: Defining constants using static so that they can be accessed without creating an object.
- **Example (Java)**:

```
class Constants {
  static final double PI = 3.14159;
}
System.out.println(Constants.PI); // Output: 3.14159
```

6. Singleton Design Pattern

- **Use Case**: In singleton design patterns, static is used to ensure that only one instance of a class is created and shared globally.
- **Example (Java)**:

```
class Singleton {
  private static Singleton instance;

  private Singleton() {} // Private constructor

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
Singleton obj1 = Singleton.getInstance();
Singleton obj2 = Singleton.getInstance();
System.out.println(obj1 == obj2); // Output: true
```

7. Static Classes in Nested Classes (C#)

- **Use Case**: In C#, you can define a static class inside another class when you want to encapsulate utility or helper methods relevant only to the enclosing class.
- **Example (C#)**:

```
class Outer {
  public static class Inner {
    public static void display() {
      Console.WriteLine("Inner Static Class");
    }
  }
}
Outer.Inner.display(); // Output: "Inner Static Class"
```

8. Static Members for Caching

- **Use Case**: Using static members to store data that is common and needs to be accessed by all instances, such as cached configuration or results.
- **Example (Java)**:

```
class Config {
  static HashMap<String, String> settings = new HashMap<>();

  static {
    settings.put("dbURL", "jdbc:mysql://localhost:3306/mydb");
  }
}
System.out.println(Config.settings.get("dbURL")); // Output: "jdbc:mysql://localhost:3306/mydb"
```

9. Static Factory Methods

- **Use Case**: Using static methods to return instances of a class, often used in place of constructors for clarity and flexibility.
- **Example (Java)**:

```
class Car {
  private String model;

  private Car(String model) {
    this.model = model;
  }
  public static Car createTesla() {
    return new Car("Tesla");
  }
}
Car car = Car.createTesla();
System.out.println(car.model); // Output: "Tesla"
```

10. Static Fields in Enum (Java)

- **Use Case**: In Java, static variables can be used inside an enum to define common properties or behaviors for all enum constants.
- **Example (Java)**:

```
enum Day {
  MONDAY, TUESDAY, WEDNESDAY;

  static final String weekType = "Work Week";
}
System.out.println(Day.MONDAY + ": " + Day.weekType); // Output: MONDAY: Work Week
```

11. Global Counters or ID Generation

- **Use Case**: static variables are commonly used to track global counters or generate unique IDs across all instances of a class.
- **Example (Java)**:

```
class User {
  static int idCounter = 0;
  int id;
  User() {
    id = ++idCounter;
  }
}
User u1 = new User();
User u2 = new User();
System.out.println(u1.id); // Output: 1
System.out.println(u2.id); // Output: 2
```

12. Restricting Class Instantiation (Static Classes)

- **Use Case**: In languages like C#, static classes cannot be instantiated, making them ideal for utility or helper methods.

- **Example (C#)**:

```
Csharp-code
public static class MathHelper {
  public static int Add(int x, int y) {
    return x + y;
  }
}
Console.WriteLine(MathHelper.Add(5, 3)); // Output: 8
```

13. Static Imports (Java)

- **Use Case**: In Java, static can be used to import static methods or fields from other classes directly, simplifying the syntax.

- **Example (Java)**:

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;
public class Main {
  public static void main(String[] args) {
    System.out.println(PI); // Output: 3.14159
    System.out.println(sqrt(16)); // Output: 4.0
  }
}
```

14. Optimizing Memory Usage for Constants

- **Use Case**: static is used for constants in languages like C++ to ensure that only one instance of the constant is stored in memory, even when multiple objects are created.

Example (C++):

```
class Circle {
  public:
    static const double PI;
};

const double Circle::PI = 3.14159;
```

These use cases illustrate how the static keyword provides powerful functionality for sharing data across class instances, managing class-level behavior, and creating efficient and structured programs in a variety of programming

Super keyword in java

The super keyword in Java is used to refer to the **parent class (superclass)** of the current object. It allows a subclass to access methods, variables, or constructors of its parent class. The super keyword is particularly useful when working with **inheritance**, where one class inherits properties and behaviour from another.

Uses of super Keyword in Java:

1. **Calling the Parent Class Constructor**:

   - The super() function is used to call the **constructor** of the parent class. This is useful when a subclass needs to initialize properties defined in the parent class.
   - If not explicitly called, the compiler automatically inserts a call to the no-argument constructor of the parent class (i.e., super()).

Example:

```
class Parent {
    Parent() {
        System.out.println("Parent class constructor called");
    }
}
class Child extends Parent {
    Child() {
        super(); // Calls Parent class constructor
        System.out.println("Child class constructor called");
    }
}

public class Test {
    public static void main(String[] args) {
        Child childObj = new Child();  // Output: Parent class constructor called
                                       // Child class constructor called
    }
}
```

- Here, the super() call in the Child constructor calls the Parent constructor first.

2. **Accessing Parent Class Methods**:

- The super keyword can be used to call a method of the parent class that is **overridden** in the subclass.
- This allows the subclass to use the parent class's version of the method, even if it has its own implementation.

Example:

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

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

    void show() {
        super.display();  // Calls the Parent class's display() method
        System.out.println("Show method in Child class");
    }
}

public class Test {
    public static void main(String[] args) {
        Child childObj = new Child();
        childObj.show();  // Output: Display method in Parent class
                          //      Show method in Child class
    }
}
```

- In this example, super.display() is used in the show() method to call the display() method of the Parent class, even though Child has its own display() method.

3. **Accessing Parent Class Variables**:

- The super keyword can also be used to access **instance variables** of the parent class, when they are hidden by variables in the subclass (i.e., when the child class has a variable with the same name as the parent class).

Example:

```
class Parent {
    int num = 100;
}
class Child extends Parent {
    int num = 200;

    void displayNumbers() {
        System.out.println("Child class num: " + num);
        System.out.println("Parent class num using super: " + super.num);
    }
}

public class Test {
    public static void main(String[] args) {
        Child childObj = new Child();
        childObj.displayNumbers();  // Output: Child class num: 200
                                    //         Parent class num using super: 100
    }
}
```

- In this example, super.num is used to access the num variable from the parent class (Parent), even though the child class (Child) has a variable with the same name.

Important Points about super:

- super() must be the **first statement** in the constructor when calling the parent class constructor.
- If the parent class does not have a no-argument constructor, the child class must call the appropriate constructor using super(arguments).
- super can be used to call **both methods and variables** from the parent class.
- **Overridden methods** can still be accessed using super.methodName() from the child class.

Example Combining All Uses of super:

```
class Parent {
    int num;

    // Parent class constructor
    Parent(int num) {
        this.num = num;
        System.out.println("Parent class constructor called");
    }
    // Parent class method
    void display() {
        System.out.println("Display method in Parent class. Num = " + num);
    }
}
class Child extends Parent {
    int num;
    // Child class constructor
    Child(int num1, int num2) {
        super(num1);  // Call to Parent class constructor
        this.num = num2;
        System.out.println("Child class constructor called");
    }

    // Overridden method
    void display() {
        super.display();  // Call to Parent class display method
        System.out.println("Display method in Child class. Num = " + num);
    }
}

public class Test {
    public static void main(String[] args) {
        Child childObj = new Child(10, 20);
        childObj.display();
    }
}
```

Output:

kotlin

Copy code

```
Parent class constructor called
Child class constructor called
Display method in Parent class. Num = 10
Display method in Child class. Num = 20
```

Summary of super:

- **Constructor Calls**: super() calls the parent class's constructor.
- **Method Calls**: super.methodName() calls the parent class's method, even if it's overridden in the child class.
- **Variable Access**: super.variableName accesses the parent class's variable when a variable with the same name exists in the child class.

The super keyword plays a crucial role in inheritance, allowing subclasses to interact with the parent class's members and ensuring the correct behavior in cases where overriding occurs.

✹Object class in java

The **Object class** is the **root superclass** for all classes. Every class in Java either directly or indirectly inherits from the Object class. If a class doesn't explicitly extend another class, it implicitly extends Object. This means all Java classes have the functionality provided by the Object class, making it the **top-most class** in the inheritance hierarchy.

Key Points about Object Class:

1. **Superclass of All Classes**: Every class in Java is a subclass of Object, either directly or indirectly.
2. **Basic Methods Available in All Classes**: The Object class provides several methods that are common to all Java objects, such as equals(), hashCode(), toString(), and clone().
3. **Can Be Used for General References**: A variable of type Object can refer to any object in Java, regardless of its class.

Important Methods of the Object Class:

1. **toString()**:
   - Returns a **string representation** of the object.
   - This method is called when you try to print an object (e.g., using System.out.println(object)).

- **By default, it returns the class name followed by the object's hash code**, but it can be **overridden** to provide a more meaningful representation of the object.

Default Implementation:

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Example:

```
class MyClass {
    int value;
    MyClass(int value) {
        this.value = value;
    }
    @Override
    public String toString() {
        return "MyClass: value = " + value;
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj = new MyClass(10);
        System.out.println(obj);  // Output: MyClass: value = 10
    }
}
```

2. **equals(Object obj)**:
    - Used to **compare two objects** for equality.
    - By default, it compares the **memory addresses** of the objects (i.e., it checks whether the two references point to the same object).
    - Typically overridden to compare the **content** of the objects, such as comparing fields/attributes.

Default Implementation:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

Example:

```
class MyClass {
    int value;

    MyClass(int value) {
        this.value = value;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        MyClass other = (MyClass) obj;
        return value == other.value;
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass(10);
        MyClass obj2 = new MyClass(10);
        System.out.println(obj1.equals(obj2));  // Output: true
    }
}
```

3. **hashCode()**:
    - Returns an **integer hash code** for the object, used for hashing-based collections like HashMap, HashSet, etc.
    - The hashCode() method should be overridden whenever equals() is overridden, ensuring that objects considered equal by equals() have the same hash code.
    - By default, it returns a hash code based on the object's memory address.

Example:

```
class MyClass {
    int value;

    MyClass(int value) {
        this.value = value;
    }

    @Override
    public int hashCode() {
        return value; // Custom hash code based on value field
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        MyClass other = (MyClass) obj;
        return value == other.value;
    }
}

public class Test {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass(10);
        MyClass obj2 = new MyClass(10);
        System.out.println(obj1.hashCode() == obj2.hashCode());  // Output: true
    }
}
```

4. **clone()**:
    - Creates and returns a **copy** of the object.

- The class must implement the Cloneable interface and override the clone() method. If the class does not implement Cloneable, a CloneNotSupportedException is thrown.
- This method performs a **shallow copy** by default (i.e., the fields of the object are copied, but if the field is a reference to an object, only the reference is copied, not the actual object).

Example:

```
class MyClass implements Cloneable {
  int value;
  MyClass(int value) {
    this.value = value;
  }
  @Override
  public MyClass clone() throws CloneNotSupportedException {
    return (MyClass) super.clone();  // Performs a shallow copy
  }
}

public class Test {
  public static void main(String[] args) throws CloneNotSupportedException {
    MyClass obj1 = new MyClass(10);
    MyClass obj2 = obj1.clone();
    System.out.println(obj1.value == obj2.value);  // Output: true
  }
}
```

5. **finalize()**:
   - This method is called by the **garbage collector** before an object is destroyed.
   - It is used to perform any cleanup actions before the object is removed from memory, but its use is generally discouraged. Modern Java programming relies on explicit resource management (e.g., try-with-resources or manual cleanup methods) rather than relying on finalize().

Note: The finalize() method is deprecated as of Java 9 and will likely be removed in future versions of Java.

6. **getClass()**:
   - Returns the **class object** of the object at runtime. It can be used to obtain metadata about the class of an object (such as its name, fields, methods, etc.).

Example:

```
class MyClass {
}
public class Test {
  public static void main(String[] args) {
    MyClass obj = new MyClass();
    System.out.println(obj.getClass().getName());  // Output: MyClass
  }
}
```

7. **wait(), notify(), and notifyAll()**:
   - These methods are used for **inter-thread communication**.
   - wait() causes the current thread to wait until another thread invokes notify() or notifyAll() on the same object.
   - notify() wakes up one thread waiting on the object's monitor, while notifyAll() wakes up all waiting threads.

Summary of Important Methods in Object Class:

Method	**Description**
toString()	Returns a string representation of the object.
equals(Object)	Checks if two objects are equal based on content (default is reference equality).
hashCode()	Returns a hash code for the object, used in hashing-based collections.
clone()	Creates and returns a copy of the object (requires implementing Cloneable).
finalize()	Called by the garbage collector before the object is destroyed.
getClass()	Returns the runtime class of the object.
wait(), notify(), notifyAll()	Methods for inter-thread communication.

Conclusion:

The Object class is fundamental in Java because it provides basic methods that are inherited by all classes. These methods can be overridden to suit the specific needs of each class, making them customizable and allowing for better functionality across Java programs.

✹Differnce between == and equals in java

In Java, the == operator and the equals() method are used for comparing objects, but they serve different purposes and behave differently. Here's a breakdown of the differences:

1. Comparison Type

- **== (Reference Comparison)**:
    - It compares **memory addresses** (references) of two objects to see if they point to the **same location** in memory.
    - For primitive types (e.g., int, char, etc.), == compares the **actual values**.
    - For objects, it checks whether two references point to the **same object instance**.
- **equals() (Content Comparison)**:
    - It is a method that compares the **content (state)** of two objects to determine if they are **meaningfully equivalent**.
    - By default, in the Object class (the superclass of all classes), equals() behaves just like ==, but it can (and often is) overridden in custom classes to compare the actual data inside the objects rather than their memory references.

2. Usage with Primitives vs. Objects

- **== with Primitives**:
    - Directly compares the value of primitive types.
    - Example:

```
int a = 5;
int b = 5;
```

```
System.out.println(a == b); // Output: true (compares values)
```

- **== with Objects**:
  - Compares the references of objects (not their contents).
  - Example:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println(s1 == s2); // Output: false (different memory references)
```

- **equals() with Objects**:
  - Typically used to compare the content (logical equality) of two objects. Many classes like String, Integer, and custom classes override the equals() method to provide meaningful comparison.

Example:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
System.out.println(s1.equals(s2)); // Output: true (compares content)
```

3. Overriding equals()

- **== cannot be overridden** as it is an operator.
- **equals()** can be overridden in user-defined classes to provide custom comparison logic.

Example:

```
class Person {
  String name;
  Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;
    Person person = (Person) obj;
    return name.equals(person.name);
  }
}

Person p1 = new Person("John");
Person p2 = new Person("John");
System.out.println(p1.equals(p2)); // Output: true (compares content)
System.out.println(p1 == p2);      // Output: false (different references)
```

4. Default Behavior

- **==**: Compares object references or primitive values. No custom logic is allowed.
- **equals()**: The default implementation (from the Object class) is identical to ==, but it can be overridden to compare object content.

5. Common Usage

- **==** is used for:
  - Primitive type comparisons (int, float, char, etc.).
  - Checking if two object references point to the same instance.

- **equals()** is used for:
    - Comparing the content or logical equality of two objects.
    - Commonly used in collection frameworks (e.g., when using HashSet or HashMap).

6. Example with String

- == compares references:

```
String str1 = new String("abc");
String str2 = new String("abc");
System.out.println(str1 == str2);     // Output: false (different objects in memory)
```

- **equals()** compares content:

```
System.out.println(str1.equals(str2)); // Output: true (same content)
```

7. Performance

- **==**:
    - Comparisons with == are faster since they are direct reference or value comparisons.
- **equals()**:
    - Might be slower if overridden, depending on the complexity of the comparison logic (e.g., deep comparison of fields).

Summary Table:

Feature	**==**	**equals()**
Comparison Type	Reference comparison	Content (or logical) comparison
Primitive Types	Compares actual values	N/A
Objects	Compares memory addresses	Compares the content (if overridden)
Overridable	No	Yes
Usage	Primitives, object references	Object content equality

In conclusion, use == when you need to compare object references or primitive values, and use equals() when you need to compare the actual content of two objects.

--OOPs--

✸ What is the need for OOPs?

There are many reasons why OOPs is mostly preferred, but the most important among them are:

- OOPs helps users to understand the software easily, although they don't know the actual implementation.
- With OOPs, the readability, understandability, and maintainability of the code increase multifold.
- Even very big software can be easily written and managed easily using OOPs.

✸ What are some major Object Oriented Programming languages?

The programming languages that use and follow the Object-Oriented Programming paradigm or OOPs, are known as Object-Oriented Programming languages. Some of the major Object-Oriented Programming languages include:

- Java
- C++
- Javascript
- Python
- PHP

And many more.

✹ What is Object-Oriented Programming Language

The Object-oriented Programming Language is the one where the code revolves around the concept of **objects**. These objects are smaller parts of the code. In simple terms, a complex program is broken into smaller parts, say objects, and executed. Various functions and operations are performed on the objects. Thus, the object is the data with unique attributes and behaviour. The blueprint of an object is called a **Class**. It is a prototype defining the variables and the methods. Further, there are four basic Object-Oriented Programming Concepts that one must know. They are **Abstraction** (the internal details are hidden and only the necessary details are available to the user), **Inheritance** (Inheriting the properties of a superclass by a subclass), **Encapsulation** (wrapping of the data and methods into a single unit), and **Polymorphism** (a function or an object taking multiple forms). **Java, Python**, and **C++** are a few examples of Object-oriented programming languages.

Why Object-Oriented Programming Language?

Here are reasons why object-oriented programming language is preferred,

- It **reduces the complexity** of the program by breaking it into objects and classes.
- The development of programs is relatively **faster** with the help of its various features. **Rich libraries** are one of the features that enable **faster development**.

  (The development of programs becomes relatively faster when using programming languages or frameworks that offer **rich libraries**. A **library** in programming is a collection of pre-written code that developers can use to perform common tasks without having to write the code from scratch.

  rich libraries speed up development by providing reusable code for common tasks, allowing developers to focus on project-specific requirements and reducing the time spent on routine coding.)
- The software is also easy to maintain and has simple updating processes.
- The three main highlighting factors include **Modularity, Extensibility, and Reusability**.

1. Modularity:

Modularity refers to breaking a large program into smaller, manageable, and self-contained components or modules, each with a specific function. These modules can be developed, tested, and maintained independently.

2. Extensibility:

Extensibility refers to the ability of a system to easily add new features or capabilities without changing existing functionality. This means the design of the software anticipates future growth and provides a clear structure for incorporating new features.

3. Reusability:

Reusability refers to the practice of using existing code or components across multiple projects or parts of the same project. Reusable code is typically designed to be generic so that it can be applied in various contexts.

✸What are some advantages of using OOPs?

- OOPs is very helpful in solving very complex level of problems.
- Highly complex programs can be created, handled, and maintained easily using object-oriented programming.
- OOPs, promote code reuse, thereby reducing redundancy.
- OOPs also helps to hide the unnecessary details with the help of Data Abstraction.
- OOPs, are based on a bottom-up approach, unlike the Structural programming paradigm, which uses a top-down approach.
- Polymorphism offers a lot of flexibility in OOPs.

✸Why is OOPs so popular?

OOPs programming paradigm is considered as a better style of programming. Not only it helps in writing a complex piece of code easily, but it also allows users to handle and maintain them easily as well. Not only that, the main pillar of OOPs - Data Abstraction, Encapsulation, Inheritance, and Polymorphism, makes it easy for programmers to solve complex scenarios. As a result of these, OOPs is so popular.

✸What is meant by the term OOPs?

OOPs refers to Object-Oriented Programming. It is the programming paradigm that is defined using objects. Objects can be considered as real-world instances of entities like class, that have some characteristics and behaviours.

✸ Are there any limitations of Inheritance?

Yes, with more powers comes more complications. Inheritance is a very powerful feature in OOPs, but it has some limitations too. Inheritance needs more time to process, as it needs to navigate through multiple classes for its implementation. Also, the classes involved in Inheritance - the base class and the child class, are very tightly coupled together. So if one needs to make some changes, they might need to do nested changes in both classes. Inheritance might be complex for implementation, as well. So if not correctly implemented, this might lead to unexpected errors or incorrect outputs.

✸ What are access specifiers and what is their significance?

Access specifiers, as the name suggests, are a special type of keywords, which are used to control or specify the accessibility of entities like classes, methods, etc. Some of the access specifiers or access modifiers include “private”, “public”, etc. These access specifiers also play a very vital role in achieving Encapsulation - one of the major features of OOPs

✸polymorphism

Polymorphism is one of the core concepts in object-oriented programming (OOP), which allows objects of different classes to be treated as instances of the same class through a shared interface. The term "polymorphism" comes from the Greek words "poly" (meaning many) and "morph" (meaning form), signifying that an object can take on many forms.

In programming, polymorphism enables a single function, method, or operator to work in different ways based on the object it operates on. There are two main types of polymorphism in Java (and most other OOP languages):

1. Compile-time Polymorphism (Static Polymorphism):

This is also known as **method overloading** or **operator overloading**. It is resolved during compile time, meaning the compiler determines which method or operation to invoke based on the arguments passed. In method overloading, multiple methods can have the same name but differ in their parameter list (either in number, type, or both).

Example of Method Overloading:

```
class Calculator {
  // Overloaded method for adding integers
  int add(int a, int b) {
    return a + b;
  }
  // Overloaded method for adding doubles
  double add(double a, double b) {
    return a + b;
  }
}
public class Main {
  public static void main(String[] args) {
    Calculator calc = new Calculator();

    System.out.println(calc.add(5, 10));      // Calls the integer version
    System.out.println(calc.add(5.5, 10.2));  // Calls the double version
  }
}
```

In this example, the add method is overloaded with two different parameter types: one for integers and one for doubles. Based on the arguments, the appropriate version is invoked.

2. Runtime Polymorphism (Dynamic Polymorphism):

This is also known as **method overriding**. It occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The actual method that gets called is determined at runtime, depending on the type of the object.

Example of Method Overriding:

```
class Animal {
  void sound() {
    System.out.println("Animal makes a sound");
  }
}
class Dog extends Animal {
  @Override
  void sound() {
    System.out.println("Dog barks");
  }
}
class Cat extends Animal {
  @Override
  void sound() {
    System.out.println("Cat meows");
  }
}
public class Main {
  public static void main(String[] args) {
    Animal myDog = new Dog(); // Dog object, referenced as Animal
    Animal myCat = new Cat(); // Cat object, referenced as Animal

    myDog.sound();  // Calls Dog's overridden sound method: "Dog barks"
    myCat.sound();  // Calls Cat's overridden sound method: "Cat meows"
    }
}
```

Here, Dog and Cat override the sound() method of the Animal class. Even though myDog and myCat are of type Animal, at runtime the JVM determines that these are Dog and Cat objects and calls the corresponding overridden methods.

Key Points of Polymorphism:

- **Method Overloading** (compile-time polymorphism) is achieved by defining multiple methods with the same name but different parameter lists.
- **Method Overriding** (runtime polymorphism) allows a subclass to provide a specific implementation of a method that is already defined in the superclass.
- Polymorphism provides flexibility and code reusability, enabling objects to interact with interfaces and base classes while the specific implementation can vary at runtime.

Advantages of Polymorphism:

- **Code Reusability**: You can write generic code that works for different types of objects.
- **Extensibility**: New classes can easily be added without altering the existing code.
- **Flexibility**: It allows for the use of interfaces and base classes to define common behavior, with specific behaviours defined in derived classes.

✸(DMD) Dynamic mehtod dispatcher in java

The **dynamic method dispatcher** is the mechanism that supports **runtime polymorphism**. It ensures that the correct method is called at runtime based on the actual object's type, rather than the reference type. This is done using **method overriding** and the concept of dynamic binding.

How It Works:

When a method is overridden in a subclass, and that method is called on a superclass reference holding a subclass object, Java uses dynamic method dispatch to determine which version of the method to execute. The method that gets executed is determined by the **runtime type** of the object, not the compile-time type of the reference variable.

Key Concepts:

- **Method Overriding**: A subclass provides its own specific implementation of a method that is already defined in its superclass.
- **Dynamic (Late) Binding**: The process of linking a method call to its actual method implementation at runtime, not at compile-time.

Important Points:

- **Only instance methods are dynamically dispatched.** Static methods, private methods, and constructors are not subject to dynamic method dispatch because they are resolved at compile time.
- **It works with method overriding, not overloading.** Overloading is resolved at compile time, while overriding (with the help of dynamic method dispatch) is resolved at runtime.

Example of Dynamic Method Dispatch:

```java
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal;  // Declare reference of type Animal

        // Dynamic dispatch at runtime based on the actual object type
        animal = new Dog();
        animal.sound();  // Calls Dog's sound method, output: "Dog barks"

        animal = new Cat();
        animal.sound();  // Calls Cat's sound method, output: "Cat meows"
    }
}
```

Explanation:

1. In this example, Dog and Cat classes both override the sound() method of the Animal class.
2. The reference variable animal is of type Animal, but at runtime, it refers to either a Dog object or a Cat object.
3. Java dynamically dispatches the method call to the correct overridden method based on the **actual object** (Dog or Cat), not based on the reference type (Animal).
4. The decision of which sound() method to execute is made **at runtime** — this is the essence of **dynamic method dispatch**.

Process of Dynamic Method Dispatch:

1. A method call is made on an object reference.
2. The compiler checks if the method exists in the reference type (compile-time checking).
3. At runtime, the JVM looks at the **actual object type** the reference points to.
4. The JVM dynamically binds the method call to the correct method implementation based on the object type.

Advantages:

- **Flexibility and Extensibility**: Dynamic method dispatch enables polymorphism, allowing programs to decide which method to call at runtime. This makes it easier to extend code by adding new classes without modifying existing code.

Dynamic Method Dispatch in Action:

The JVM uses a technique called **vtable (virtual method table)** to implement dynamic method dispatch. Each class that has overridden methods maintains a vtable, which maps method calls to the actual method implementations in the class. When a method is invoked, the JVM looks up the vtable of the object's runtime class to find the correct method to call.

✹Early and late binding in java

In Java, **early binding** and **late binding** refer to when the method to be executed is determined—either at compile time or runtime. Here's a breakdown of the two concepts:

1. Early Binding (Static Binding)

- **Definition**: The method to be invoked is determined at **compile time**.
- **Occurs in**:
    - **Method Overloading**: When a class has multiple methods with the same name but different parameter lists, the compiler resolves which method to call based on the method signature.
    - **Final Methods:** Methods marked as final cannot be overridden, so they are bound early.
    - **Static Methods:** These methods belong to the class rather than an instance, and are resolved at compile time.
    - **Private Methods:** Since private methods cannot be overridden, they are also bound early.
- **Example**:

```
class Animal {
    public static void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    public static void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.sound();  // This will call Animal's sound() method due to early binding.
    }
}
```

2. Late Binding (Dynamic Binding)

- **Definition**: The method to be invoked is determined at **runtime** based on the actual object being referenced, not the reference type.
- **Occurs in**:
    - **Method Overriding:** When a subclass overrides a method from its superclass, the method that is executed depends on the actual object type (not the reference type) at runtime.

- **Example**:

```
class Animal {
   public void sound() {
      System.out.println("Animal makes a sound");
   }
}
class Dog extends Animal {
   @Override
   public void sound() {
      System.out.println("Dog barks");
   }
}

public class Main {
   public static void main(String[] args) {
      Animal animal = new Dog();  // The actual object is of type Dog
      animal.sound();  // This will call Dog's sound() method due to late binding.
   }
}
```

Key Differences

- **Early Binding**: Occurs at compile time. Used for static, final, and private methods, as well as method overloading.
- **Late Binding**: Occurs at runtime. Used for method overriding where the JVM resolves the method call based on the actual object type.

In essence, **early binding** is faster because the method to be invoked is known at compile time, while **late binding** provides flexibility and supports polymorphism but adds a slight runtime overhead.

Both overloading and overriding does not apply to instance variable. Explain?

Overloading and overriding are concepts that apply to methods, not instance variables. Here's why neither applies to instance variables:

Overloading:

Overloading allows multiple methods with the same name but different parameter lists (either in number, types, or both) to coexist within a class.

However, overloading is not applicable to instance variables because instance variables do not have parameters. An instance variable is simply a property of a class, not a function that can have multiple signatures.

Overriding:

Overriding occurs in inheritance, where a subclass provides a specific implementation of a method that is already defined in its parent class. It allows runtime polymorphism.

Instance variables cannot be overridden because they are not subject to dynamic dispatch. They are resolved at compile-time based on the reference type, not the actual object's type at runtime. Even if an instance variable with the same name exists in a subclass, it will shadow the parent class's variable, but this is not considered overriding.

In short:

Overloading: Instance variables can't have multiple signatures or types.

Overriding: Instance variables are resolved at compile-time, and their access depends on the reference type, not overridden at runtime.

Encapsulation java

Encapsulation is one of the four fundamental principles of object-oriented programming (OOP), and in Java, it refers to the concept of wrapping or bundling the data (variables) and the methods (functions) that operate on that data into a single unit, typically a class. Encapsulation restricts direct access to some of an object's components and ensures that the internal representation of an object is hidden from the outside world. This is done by making the variables private and providing public getter and setter methods to access or modify them.

Key Concepts of Encapsulation:

1. **Data Hiding**: The object's state (its data) is hidden from the outside, which means the internal details of an object are hidden, and only necessary data is exposed.
2. **Controlled Access**: Access to the object's data is controlled through methods (getters and setters). This allows you to control how the data is modified or accessed.
3. **Increased Security**: By preventing direct access to the data, encapsulation helps secure the object's state and protects it from unwanted changes.

How Encapsulation is Achieved:

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.

1. **Private Variables**: The data fields (variables) of a class are marked as private, so they cannot be accessed directly from outside the class.
2. **Public Methods**: Public methods (getters and setters) are used to read or update the values of private variables.

Example of Encapsulation:

```
class Person {
    // Private data members
    private String name;
    private int age;

    // Public getter for 'name'
    public String getName() {
        return name;
    }

    // Public setter for 'name'
    public void setName(String name) {
        this.name = name;
    }

    // Public getter for 'age'
    public int getAge() {
        return age;
    }

    // Public setter for 'age'
    public void setAge(int age) {
        // You can add validation logic here
        if (age > 0) {
            this.age = age;
        } else {
            System.out.println("Age must be positive");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of the Person class
        Person person = new Person();

        // Setting the values of private fields using setters
        person.setName("John");
        person.setAge(25);

        // Accessing the values using getters
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
}
```

Advantages of Encapsulation:

- By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.
- It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.
- It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.
- The encapsulate class is **easy to test**. So, it is better for unit testing.
- The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

1. **Improved Security**: Since the variables are private, external classes cannot directly access or modify them, ensuring data integrity.
2. **Controlled Modification**: The setter methods allow controlled changes to the fields, enabling validation or logging when necessary.
3. **Code Maintenance**: Encapsulation makes the code more modular. If changes are needed, you can modify the internals of the class without affecting other parts of the code.

4. **Increased Flexibility**: Getters and setters allow changing the internal representation of the data without modifying the external interface (public methods).

Conclusion:

Encapsulation is a core principle in Java's OOP that promotes better data management, security, and modular design. It hides the internal implementation details of an object while exposing only what is necessary to interact with the object externally.

✹ Abstraction in java

Abstraction in Java is an Object-Oriented Programming (OOP) concept that focuses on **hiding the complex implementation details** and exposing only the essential features of an object or system. It allows the user to interact with the object at a higher level, without needing to understand the internal workings of the object. In Java, abstraction is achieved through **abstract classes** and **interfaces**.

Key Points of Abstraction:

- **What it does**: Abstraction hides the internal implementation of a feature or a function and only shows the necessary details (like method signatures) to the user.
- **Why it's useful**: It simplifies the usage of objects by focusing only on what they do, not how they do it. This improves code readability, maintainability, and scalability.
- **How it's achieved**: In Java, abstraction is implemented using:
    - **Abstract Classes**: These classes cannot be instantiated directly and can contain both abstract methods (without a body) and concrete methods (with an implementation).
    - **Interfaces**: They provide complete abstraction by defining methods without any implementations. The implementing class must provide the method bodies.

Example of Abstraction Using an Abstract Class:

An abstract class can have both abstract methods (methods without implementations) and concrete methods (methods with implementations).

Note:

- We can't create a abstract static method().
- Also, we can't create object of an abstract class.

```
// Abstract class
abstract class Animal {
    // Abstract method (without implementation)
    public abstract void sound();

    // Concrete method (with implementation)
    public void sleep() {
        System.out.println("The animal is sleeping.");
    }
}
// Subclass provides the implementation of the abstract method
class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

```java
class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();  // Polymorphism
        animal.sound();  // Output: Dog barks
        animal.sleep();  // Output: The animal is sleeping.
    }
}
```

- **Explanation**: In this example, the Animal class is abstract, and it defines an abstract method sound(). The subclass Dog provides the actual implementation for the sound() method. The sleep() method in the Animal class is concrete, so it can be used directly by any subclass.

Example of Abstraction Using an Interface:

Interfaces in Java provide **100% abstraction**, meaning all methods defined in an interface are abstract (until Java 8, when default and static methods were introduced).

```java
// Interface
interface Vehicle {
    // All methods in an interface are abstract by default
    void start();
    void stop();
}
// Class implementing the interface
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car is starting.");
    }
    @Override
    public void stop() {
        System.out.println("Car is stopping.");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car();  // Polymorphism
        car.start();  // Output: Car is starting.
        car.stop();   // Output: Car is stopping.
    }
}
```

- **Explanation**: The Vehicle interface defines two abstract methods, start() and stop(). The Car class implements this interface and provides the implementations for these methods.

Levels of Abstraction:

1. **Abstract Class**:
    - Can contain both abstract and concrete methods.
    - Supports partial abstraction (some methods may have an implementation).
    - Allows constructor and instance variables.
2. **Interface**:
    - Provides full abstraction.
    - Can only contain abstract methods (before Java 8) and static or default methods (since Java 8).
    - Does not allow constructors or instance variables.

When to Use Abstract Class vs Interface:

- **Abstract Class**: Use when you have a common base class with some shared implementation that other subclasses can inherit.

  An **abstract class** is used when you want to define some **common behaviour** for all subclasses but leave some methods unimplemented (to be defined in the subclasses). An abstract class can have a mixture of **fully implemented methods**, **abstract methods** (methods with no body), and **instance variables**.

- **Interface**: Use when you want to specify a contract for behaviour without any implementation, ensuring that all implementing classes follow the same set of rules.

  An **interface** is a completely abstract class that is used to define a **contract** that other classes must follow. An interface defines **what** methods a class should implement, but not **how**. Interfaces are more flexible than abstract classes because a class can implement multiple interfaces, allowing for a form of multiple inheritance.

Advantages of Abstraction:

1. **Reduces complexity**: By hiding the details, it allows programmers to focus on the "what" rather than the "how."
2. **Improves maintainability**: Changing the implementation in an abstract class or interface doesn’t affect the code that uses the abstraction.
3. **Promotes reusability**: Interfaces allow for better code reuse as multiple classes can implement the same interface.

Summary:

In Java, **abstraction** is the technique of hiding implementation details and exposing only the necessary parts of a class or object. It can be achieved through **abstract classes** (partial abstraction) or **interfaces** (full abstraction), allowing users to interact with objects at a higher level of abstraction. This leads to more readable and maintainable code.

✸ What is an interface?

An interface refers to a special type of class, which contains methods, but not their definition. Only the declaration of methods is allowed inside an interface. To use an interface, you cannot create objects. Instead, you need to implement that interface and define the methods for their implementation.

✸ Difference between encapsulation and abstraction

Encapsulation and abstraction are both fundamental principles of object-oriented programming (OOP) in Java, but they serve different purposes. Here's a breakdown of the key differences between the two:

1. Encapsulation:

- **Definition**: Encapsulation is the practice of bundling data (variables) and methods that manipulate that data into a single unit (class), while restricting access to the inner workings of the class.
- **Goal**: Its main goal is **data hiding** and controlling access to the internal state of an object through public methods (getters and setters).
- **How it works**: In encapsulation, class fields are typically marked as private, and access is provided through public methods.

- **Purpose**: It focuses on protecting the internal state of the object and controlling how external code interacts with it.

- **Example**:

```
class Employee {
   private String name;  // Private data member (hidden from outside)
   private double salary;

   public String getName() {  // Public getter method
      return name;
   }

   public void setName(String name) {  // Public setter method
      this.name = name;
   }

   public double getSalary() {
      return salary;
   }

   public void setSalary(double salary) {
      if(salary > 0) {
         this.salary = salary;
      } else {
         System.out.println("Salary cannot be negative.");
      }
   }
}
```

  - In this example, the internal state of the Employee class is encapsulated, as direct access to name and salary is restricted.

2. Abstraction:

- **Definition**: Abstraction is the concept of hiding the complex implementation details and showing only the essential features or functionality. It provides a simple interface for users to interact with the system.
- **Goal**: Its main goal is to focus on **what** an object does, rather than **how** it does it.
- **How it works**: Abstraction can be achieved using abstract classes and interfaces. Users of the abstract class or interface only know the method signatures, not their implementations.
- **Purpose**: It simplifies the interaction by only exposing the necessary details while hiding the internal complexity of how things work.

Example:

```
abstract class Vehicle {  // Abstract class
   abstract void start();  // Abstract method (no implementation)
   abstract void stop();

   public void fillFuel() {
      System.out.println("Fueling the vehicle.");
   }
}

class Car extends Vehicle {
   @Override
   public void start() {  // Providing implementation of the abstract method
      System.out.println("Car is starting.");
```

```
    }

    @Override
    public void stop() {
        System.out.println("Car is stopping.");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start();  // Abstracted interaction
        car.fillFuel();
        car.stop();
    }
}
```

- In this example, the user interacts with the Car object through the Vehicle class, without knowing the internal implementation of start() and stop() methods.

Key Differences Between Encapsulation and Abstraction:

Aspect	**Encapsulation**	**Abstraction**
Definition	Wrapping data (variables) and methods into a single unit (class) and restricting access to some components.	Hiding the internal implementation details and showing only the essential features.
Purpose	To **hide the internal state** and protect object data from external modification.	To **simplify complexity** by exposing only essential details while hiding the implementation.
Focus	**How** to protect and control access to data.	**What** an object does, not how it does it.
How it's achieved	Using access modifiers like private, protected, and public getter/setter methods.	Using abstract classes, interfaces, and abstract methods.
Example	Private fields with public getter/setter methods.	Abstract classes or interfaces where details are hidden from the user.
Level of Control	Provides controlled access to data.	Provides a high-level interface without showing the implementation details.
Real-world analogy	A capsule that hides its internal contents.	A TV remote that shows only buttons, hiding the internal electronics.

Summary:

- **Encapsulation** focuses on **data protection** and restricting access to an object's internals, ensuring controlled interaction.
- **Abstraction** focuses on **simplifying complexity** by showing only the necessary parts of an object and hiding the intricate details of its implementation.

Both concepts work together to create a clean, secure, and manageable design in object-oriented programming.

encapsulation is data hiding and abstraction is implementation hiding is this correct

Yes, you are correct! The distinction you've made between **encapsulation** and **abstraction** is accurate:

1. **Encapsulation** is primarily about **data hiding**.
    - It restricts direct access to the internal data of an object by making the data private and only accessible through controlled methods (like getters and setters).
    - The focus is on **protecting the internal state** of an object from unauthorized access or modification.
    - Example: Private variables in a class that can only be accessed or modified via public methods.
2. **Abstraction** is about **implementation hiding**.
    - It hides the complexity of the internal implementation and only exposes essential details (what the object does, not how it does it).
    - The focus is on providing a clear, simple interface for interacting with an object or system, without exposing the underlying code or logic.
    - Example: Abstract classes or interfaces where the user doesn't know or need to know how the methods are implemented, only that they exist and can be used.

In summary:

- **Encapsulation** hides the **data**.
- **Abstraction** hides the **implementation**.

Both are critical concepts in object-oriented programming but serve different purposes.

Abstract class can provide the implementation of interface but interface cannot provide implementation of abstract class.

In Java, your statement is largely correct and highlights the key difference between **abstract classes** and **interfaces** when it comes to inheritance and implementation.

1. Abstract Classes Providing Implementation of Interfaces:

An **abstract class** can implement an interface and provide concrete implementations of some or all of the methods declared in the interface. Since abstract classes can contain both abstract and concrete methods, they have the flexibility to either implement some of the interface's methods or leave them abstract, which must then be implemented by a subclass.

Example:

```
// Defining an interface
interface Animal {
    void sound();
    void sleep();
}

// Abstract class implementing the interface
abstract class Dog implements Animal {
    // Providing implementation for sleep() method from the interface
    public void sleep() {
        System.out.println("Dog is sleeping.");
    }

```

```
  // sound() is not implemented, so it's still abstract
}

// Concrete class extending the abstract class
class Bulldog extends Dog {
  @Override
  public void sound() {
    System.out.println("Bulldog barks.");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog dog = new Bulldog();
    dog.sound();  // Output: Bulldog barks.
    dog.sleep();  // Output: Dog is sleeping.
  }
}
```

- **Explanation**:
  - The interface Animal declares two abstract methods, sound() and sleep().
  - The abstract class Dog implements the Animal interface and provides the implementation for the sleep() method, but it leaves the sound() method abstract, requiring any subclass (e.g., Bulldog) to provide the implementation.
  - This demonstrates that abstract classes can implement an interface and can selectively implement its methods.

2. Interfaces Cannot Provide Implementations of Abstract Classes:

An **interface** in Java cannot extend an abstract class. This is because interfaces are designed to provide full abstraction, meaning they cannot have any concrete methods (before Java 8). Even after Java 8, when default and static methods were introduced, an interface still cannot extend an abstract class because they serve different purposes.

- **Interface**: It defines a **contract** that classes must follow, but it does not define any common state or implementation (with the exception of default and static methods introduced in Java 8).
- **Abstract Class**: It is a **partially implemented class** that can provide some default behavior and shared state (fields), while still leaving some methods abstract.

Why Interfaces Cannot Provide Implementation of Abstract Classes:

1. **Design Philosophy**: Interfaces define pure behavior and act as a contract for classes to implement. An abstract class, on the other hand, may contain actual fields and concrete methods, which an interface cannot handle because interfaces are meant to work independently of any particular implementation.
2. **Fields**: Abstract classes can have fields (state), but interfaces cannot (except constants). This is a fundamental difference. An interface cannot contain or inherit implementation details, like fields or non-default methods from an abstract class.
3. **Inheritance**: Abstract classes are meant to be part of an inheritance hierarchy where they share common functionality among subclasses. Interfaces, on the other hand, are intended for providing multiple types of behaviours that can be implemented by any class.

Example (Not Allowed):

```
abstract class Animal {
  abstract void sound();
  void eat() {
    System.out.println("Eating...");
    }
}

// This is illegal in Java. An interface cannot extend an abstract class.
interface Dog extends Animal {
    // No method can be implemented from the abstract class.
}
```

Key Differences Between Abstract Classes and Interfaces:

Feature	**Abstract Class**	**Interface**
Implementation	Can have both abstract methods and concrete methods.	Can have only abstract methods (before Java 8) or default/static methods (after Java 8).
State (Fields)	Can have instance variables (state).	Cannot have instance variables (except constants).
Constructors	Can have constructors.	Cannot have constructors.
Inheritance	Can extend only one class (single inheritance).	Can extend multiple interfaces (multiple inheritance of behavior).
Implements	Can implement multiple interfaces.	Cannot implement an abstract class.

Summary:

- **Abstract Classes** can implement interfaces and provide implementations for some or all of the methods, while also having abstract methods to be implemented by subclasses.
- **Interfaces** cannot extend abstract classes or provide implementations of abstract classes because they are designed to provide only method signatures (and default/static methods since Java 8) without holding state or concrete behavior.

This distinction arises from their different roles in Java:

- **Abstract classes** serve as a common base with shared behavior, partially implemented.
- **Interfaces** define a contract for classes to follow, specifying what they must do but not how they do it.

✹ What is Inheritance

Inheritance is one of the major features of object-oriented programming, by which an entity inherits some characteristics and behaviors of some other entity and makes them their own. Inheritance helps to improve and facilitate code reuse.

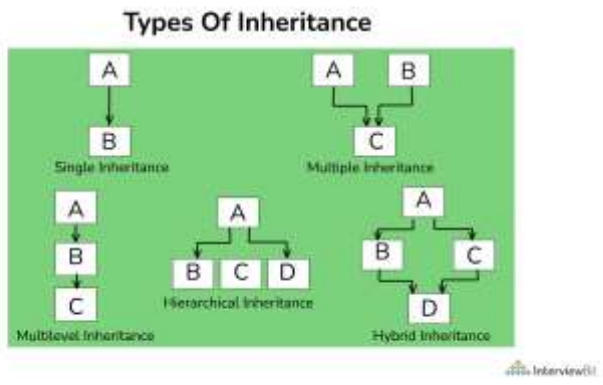
Let me explain to you with a common example. Let's take three different vehicles - a car, truck, or bus. These three are entirely different from one another with their own specific characteristics and behavior. But. in all three, you will find some common elements, like steering wheel, accelerator, clutch, brakes, etc. Though these elements are used in different vehicles, still they have their own features which are common among all vehicles. This is achieved with inheritance. The car, the truck, and the bus have all inherited the features like steering wheel, accelerator, clutch, brakes, etc, and used them as their own. Due to this, they did not have to create these components from scratch, thereby facilitating code reuse.

✹ What are the various types of inheritance?

The various types of inheritance include:

- Single inheritance

- Multiple inheritances
- Multi-level inheritance
- Hierarchical inheritance
- Hybrid inheritance

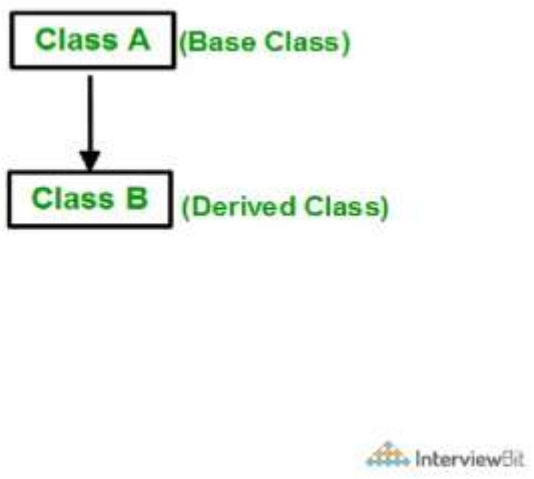


✹ What is a subclass?

The subclass is a part of Inheritance. The subclass is an entity, which inherits from another class. It is also known as the child class.

✹ Define a superclass?

Superclass is also a part of Inheritance. The superclass is an entity, which allows subclasses or child classes to inherit from itself.



✹ Java does not support multiple inheritance

- Java does not support **multiple inheritance** through classes to avoid the complexity and ambiguity that can arise from it. Multiple inheritance refers to a feature where a class can inherit from more than one superclass. While it can be useful in some cases, it also introduces several problems, especially with conflicting methods or fields from different parent classes.
- Here's a detailed explanation of why Java does not support multiple inheritance:

1. **Diamond Problem**

- One of the major reasons Java avoids multiple inheritance is the **diamond problem** (also known as the **ambiguity problem**). The diamond problem occurs when a class inherits from two classes that have a common ancestor, leading to ambiguity.

2. **Simplicity and Avoiding Complexity**
3. **Preserving Maintainability and Readability**
4. **Java's Alternative: Interfaces**

   Java solves the problem of multiple inheritance by providing interfaces. An interface allows a class to inherit behavior from multiple sources, without the complexity of multiple inheritance from classes. A class can implement multiple interfaces, which lets it inherit method signatures (but not the method implementation).

✹ Exception handling in java

Exception Handling in Java is a powerful mechanism that handles runtime errors, allowing the normal flow of the program to continue. It ensures that when an exception (an event that disrupts the normal flow of a program) occurs, it is managed gracefully, preventing the program from crashing.

Key Concepts in Exception Handling:

1. **Exception**: An exception is an abnormal condition that occurs during the execution of a program. Exceptions in Java are objects representing an error. There are two types:
   - **Checked Exceptions**: These are exceptions that are checked at compile-time, like IOException, SQLException. The compiler forces the programmer to handle these exceptions.
   - **Unchecked Exceptions (Runtime Exceptions)**: These occur at runtime and are not checked at compile-time. Examples include NullPointerException, ArrayIndexOutOfBoundsException.
   - **Errors**: These represent serious issues that usually cannot be recovered from, such as OutOfMemoryError, StackOverFlowError.
2. **Exception Hierarchy**:
   - All exceptions in Java are subclasses of the Throwable class.
   - Throwable has two direct subclasses:
     - Error: Represents errors in the system (e.g., memory issues, JVM errors) that usually cannot be handled by the application.
     - Exception: Represents recoverable conditions. The Exception class has two main branches:
       - **Checked Exceptions**: Subclasses of Exception that must be caught or declared in the method.
       - **Unchecked Exceptions**: Subclasses of RuntimeException.

java.lang.Object
↳ java.lang.Throwable
↳ java.lang.Error
↳ java.lang.Exception
↳ java.lang.RuntimeException

3. **Basic Syntax of Exception Handling**: Java provides five keywords for handling exceptions:
   - **try**: The block of code that might throw an exception.

- **catch**: The block of code that catches and handles the exception.
- **finally**: The block of code that is always executed, regardless of whether an exception occurred or not (used for cleanup activities).
- **throw**: Used to explicitly throw an exception.
- **throws**: Declares exceptions that a method might throw.

Example of Exception Handling:

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            // Code that may throw an exception
            int data = 10 / 0;  // ArithmeticException (division by zero)
            System.out.println(data);
        } catch (ArithmeticException e) {
            // Handling the exception
            System.out.println("Cannot divide by zero: " + e.getMessage());
        } finally {
            // This block is always executed
            System.out.println("Finally block executed.");
        }
        System.out.println("Rest of the program continues...");
    }
}
```

Output:

```
Cannot divide by zero: / by zero
Finally block executed.
Rest of the program continues...
```

- **Explanation**:
  - In the try block, an ArithmeticException is thrown due to division by zero.
  - The catch block catches this exception and handles it by printing an error message.
  - The finally block is executed regardless of whether an exception occurs or not.
  - The program does not crash and continues executing the next statement after the try-catch-finally block.

Common Scenarios in Exception Handling:

1. **Multiple Catch Blocks**: You can handle multiple exceptions by providing separate catch blocks for each exception type.

```
try {
    // code that may throw multiple exceptions
} catch (IOException e) {
    System.out.println("IO error occurred");
} catch (SQLException e) {
    System.out.println("SQL error occurred");
}
```

2. **Catching Multiple Exceptions in a Single Catch Block** (Java 7+): You can catch multiple exceptions in a single catch block by using the pipe (|) operator.

```
try {
    // code that may throw IOException or SQLException
} catch (IOException | SQLException e) {
    System.out.println("Error occurred: " + e.getMessage());
}
```

3. **The finally Block**: The finally block is optional and is used to execute important code such as closing resources like files or database connections, whether an exception is handled or not.

```
try {
   // open a resource
} catch (Exception e) {
   System.out.println("An exception occurred");
} finally {
   // close the resource (cleanup)
   System.out.println("Resource closed.");
}
```

4. **Throwing Exceptions**: You can throw exceptions manually using the throw keyword.

```
public void validateAge(int age) {
   if (age < 18) {
      throw new ArithmeticException("Not eligible to vote");
   } else {
      System.out.println("Eligible to vote");
   }
}
```

5. **Declaring Exceptions**: If a method might throw a checked exception, it must declare it using the throws keyword.

```
public void readFile() throws IOException {
   FileReader file = new FileReader("test.txt");
   file.read();
}
```

6. **Custom Exceptions**: You can create your own exceptions by extending the Exception class or the RuntimeException class.

```
// Creating a custom exception
class InvalidAgeException extends Exception {
   public InvalidAgeException(String message) {
      super(message);
   }
}

// Using the custom exception
public class TestCustomException {
   public static void main(String[] args) {
      try {
         validateAge(15);
      } catch (InvalidAgeException e) {
         System.out.println("Exception caught: " + e.getMessage());
      }
   }

   static void validateAge(int age) throws InvalidAgeException {
      if (age < 18) {
         throw new InvalidAgeException("Age is less than 18");
      } else {
         System.out.println("Valid age");
      }
   }
}
```

Advantages of Exception Handling:

1. **Maintains Program Flow**: It allows the program to continue running even after an error occurs, rather than crashing.
2. **Separates Error Handling Logic**: By handling exceptions separately from the main logic, it keeps the code clean and readable.
3. **Propagates Errors**: Methods can declare exceptions and allow them to propagate up the call stack until they are caught by an appropriate catch block.
4. **Custom Exceptions**: You can define your own exception types, allowing you to handle specific application errors more clearly.

Exception Propagation:

When an exception is thrown, it is propagated up the method call stack until it is caught or reaches the JVM, which terminates the program. Checked exceptions must either be caught or declared in the method signature using throws.

Summary:

- **Exception handling** in Java ensures that runtime errors do not crash the program, allowing it to continue execution.
- The main keywords used in exception handling are try, catch, finally, throw, and throws.
- Exception handling promotes cleaner, more maintainable, and resilient code by isolating error-handling logic from the main code.

✹ What is the output of the following program?

```java
public class Test
{
    public static void main(String[] args)
    {
        try
        {
            System.out.printf("1");
            int data = 5 / 0;
        }
        catch(ArithmeticException e)
        {
            Throwable obj = new Throwable("Sample");
            try
            {
                throw obj;
            }
            catch (Throwable e1)
            {
                System.out.printf("8");
            }
        }
        finally
        {
            System.out.printf("3");
        }
        System.out.printf("4");
    }
}
```

- a) Compilation error
  b) Runtime error
  c) 1834
  d) 134
- **Ans.** (c)
  Explanation: Exceptions can be thrown in catch clause. This is done in order to change the exception type at run time. Exceptions in catch clause are thrown by creating instances of class Throwable as shown in the program.

✹ Condition where finally block can't execute.

Yes, while the finally block is generally guaranteed to execute in Java, there are a few rare scenarios where the finally block may not execute. These situations usually involve extreme cases such as JVM termination, system errors, or halting the program. Here are the key conditions:

1. When the JVM Exits (System.exit()):

If **System.exit()** is called in the try or catch block, it **terminates the JVM,** and the finally block will not execute because the program is halted immediately.

```
public class Test {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            System.exit(0);  // JVM terminates here
        } finally {
            System.out.println("Inside finally block");
        }
    }
}
```

Output:

```
Inside try block
```

- The finally block does **not** execute because System.exit(0) shuts down the JVM.

2. When a Fatal JVM Error Occurs:

In cases where there is a fatal JVM error (like OutOfMemoryError or StackOverflowError), the finally block may not execute because the JVM is no longer able to continue processing.

```
public class Test {
    public static void main(String[] args) {
        try {
            // Simulating a fatal error
            throw new OutOfMemoryError();
        } finally {
            System.out.println("This may not print if JVM crashes");
        }
    }
}
```

- If the JVM crashes due to the OutOfMemoryError, the finally block may not execute.

3. Thread Death (e.g., Thread.stop()):

If a thread is forcibly stopped (for instance, by calling the deprecated **Thread.stop()** method), the finally block may not execute because the thread's execution is immediately terminated.

```
public class Test {
    public static void main(String[] args) {
        try {
            Thread.currentThread().stop();  // Forcibly stops the thread
        } finally {
            System.out.println("Finally block");
        }
    }
}
```

- Since Thread.stop() forcibly stops the thread, the finally block may not be executed.

4. Power Failure or Hardware Shutdown:

If the machine running the JVM suddenly loses power or is forcibly turned off, the finally block obviously cannot execute because the entire system is shut down.

5. Infinite Loop in try or catch Block:

If an infinite loop occurs in the try or catch block, the finally block will never be reached.

```
public class Test {
    public static void main(String[] args) {
        try {
            while (true) {
                System.out.println("Infinite loop");
            }
        } finally {
            System.out.println("Finally block");  // This will never be reached
        }
    }
}
```

- Since the loop is infinite, the finally block will never get a chance to execute.

6. Forceful Termination with kill -9 (in Unix/Linux):

If you forcefully terminate a Java process using commands like kill -9 in Unix/Linux systems, the JVM is killed immediately, and the finally block will not execute.

Summary:

In most normal cases, the finally block **will execute** after the try or catch blocks. However, under extreme circumstances like calling System.exit(), encountering a JVM crash, or forcibly terminating a thread or process, the finally block will not execute.

✹ Shallow Copying and Deep Copying in Java

When copying an object in Java, there are two main types of copies: **shallow copy** and **deep copy**. These two types of copying determine how the object's properties (including references to other objects) are copied.

---

1. Shallow Copy:

In **shallow copying**, a new object is created, and the fields of the original object are copied to this new object **as-is**. If the object has fields that reference other objects, the **references** to those objects are copied (not the actual objects). As a result, both the original and copied objects will reference the same objects in memory.

Key Characteristics:

- **Primitive types**: Primitive fields (like int, char, boolean) are copied **by value**.
- **Object references**: Object fields are copied **by reference**, meaning both the original and copied objects will share the same referenced objects.

Example of Shallow Copy:

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

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

class Student implements Cloneable {
    int id;
    Person person;

    Student(int id, Person person) {
        this.id = id;
        this.person = person;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();  // Shallow copy
    }
}

public class ShallowCopyExample {
    public static void main(String[] args) throws CloneNotSupportedException {
        Person person1 = new Person("John", 25);
        Student student1 = new Student(101, person1);

        // Shallow copy of student1
        Student student2 = (Student) student1.clone();

        // Modify the person object in student2
        student2.person.name = "Alice";

        // Changes reflect in both student1 and student2
        System.out.println("student1 Person Name: " + student1.person.name);  // Alice
        System.out.println("student2 Person Name: " + student2.person.name);  // Alice
    }
}
```

Explanation:

- Both student1 and student2 reference the same Person object because shallow copying only copies the reference, not the object itself.
- If you modify the Person object in student2, it also affects student1 since both point to the same object.

---

2. Deep Copy:

In **deep copying**, a new object is created, and all fields of the original object are copied. If the object contains references to other objects, **new instances** of those referenced objects are also created (i.e., the entire object graph is copied). This ensures that the original and copied objects do not share any references to the same objects in memory.

Key Characteristics:

- **Primitive types**: Primitive fields are copied **by value**.
- **Object references**: Object fields are recursively copied, meaning **new instances** of the referenced objects are created.

Example of Deep Copy:

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

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

    // Deep copy constructor
    Person(Person person) {
        this.name = person.name;
        this.age = person.age;
    }
}

class Student {
    int id;
    Person person;

    Student(int id, Person person) {
        this.id = id;
        this.person = person;
    }

    // Deep copy constructor
    Student(Student student) {
        this.id = student.id;
        this.person = new Person(student.person);  // Create a new Person object (deep copy)
    }
}

public class DeepCopyExample {
    public static void main(String[] args) {
        Person person1 = new Person("John", 25);
        Student student1 = new Student(101, person1);

        // Deep copy of student1
        Student student2 = new Student(student1);

        // Modify the person object in student2
        student2.person.name = "Alice";

        // Changes do not reflect in student1
        System.out.println("student1 Person Name: " + student1.person.name);  // John
        System.out.println("student2 Person Name: " + student2.person.name);  // Alice
    }
}
```

Explanation:

- In this example, student1 and student2 have their own separate Person objects.
- Modifying the Person object in student2 does not affect student1 because a deep copy creates a new instance of Person for student2.

Differences Between Shallow Copy and Deep Copy:

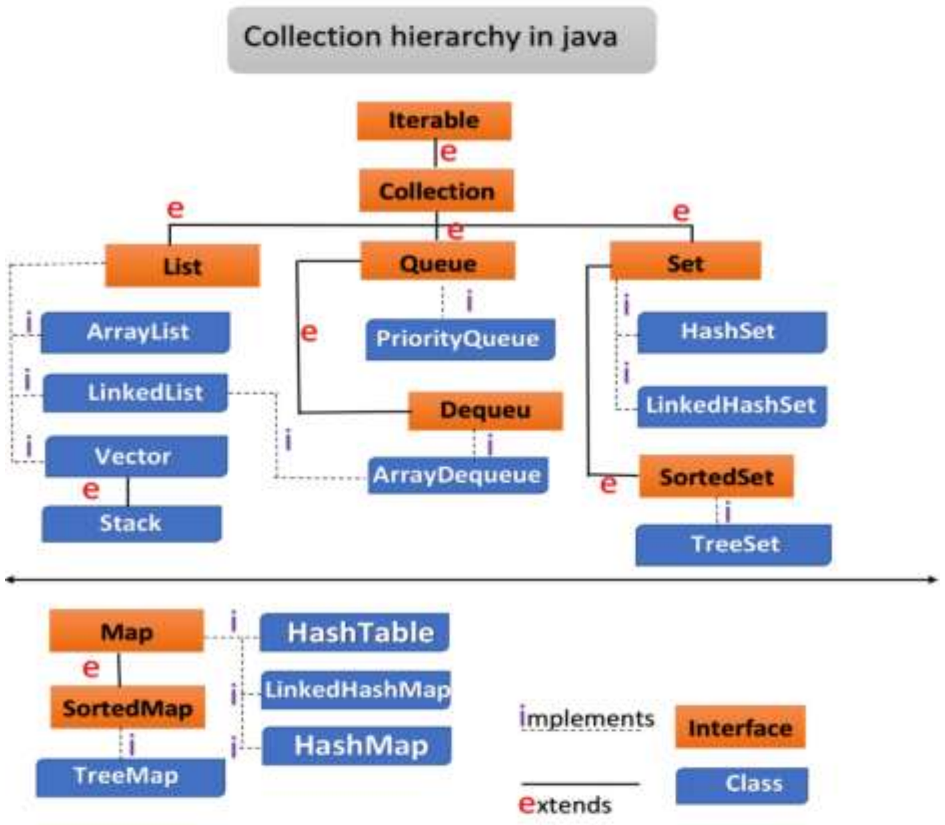
Aspect	**Shallow Copy**	**Deep Copy**
Object references	Copies **references** to the objects, not the objects themselves.	Copies the **entire object graph**, including referenced objects.
Memory allocation	The copied object shares referenced objects with the original.	New instances are created for all referenced objects.
Modification effect	Modifying a referenced object in one copy affects the other.	Modifying a referenced object in one copy does not affect the other.
Copying complexity	Generally faster and requires less memory, but can lead to unintended consequences due to shared references.	More expensive in terms of time and memory, but safer in ensuring independence of the objects.
Use case	Suitable when the original and copied objects are supposed to share common data.	Suitable when complete independence of the copied object is required.

When to Use Shallow Copy vs Deep Copy:

- **Shallow Copy** is useful when the object contains many fields that do not need deep copying, and performance or memory overhead is a concern.
- **Deep Copy** is required when you want to ensure complete separation between the original and copied objects, such as in multi-threaded environments or cases where modification of one object should not affect the other.

✹ What is a Framework in Java?

A framework is a set of classes and interfaces which provide a ready-made architecture. In order to implement a new feature or a class, there is no need to define a framework. However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.



✸ collection in java with internal implementation and use cases

Java Collections Framework provides a set of interfaces, classes, and algorithms to store and manipulate groups of objects. It includes various data structures like lists, sets, maps, and queues. Each collection class offers different ways to store and access data based on specific needs.

✸ Priority queue

A **PriorityQueue** in Java is a type of queue where each element is associated with a priority. When you retrieve an element from the queue, it always returns the element with the highest (or lowest) priority, depending on how the priority is defined. It is part of the java.util package.

Unlike regular queues that follow **First-In-First-Out (FIFO)** order, a priority queue follows **priority-based ordering**.

Key Features:

1. **Heap-based Implementation**: A priority queue in Java is implemented using a binary heap. By default, it is a **min-heap**, meaning the element with the **lowest value** has the highest priority and is dequeued first.
2. **Custom Ordering**: You can define a custom comparator to convert the priority queue into a **max-heap**, where the element with the **highest value** is dequeued first.

Constructors:

- PriorityQueue(): Creates a default min-heap.
- PriorityQueue(int initialCapacity): Creates a priority queue with the specified initial capacity.
- PriorityQueue(Comparator<? super E> comparator): Creates a priority queue with a custom comparator to define the ordering of the elements.

Methods:

1. add(E e): Adds an element e to the queue based on its priority.

2. peek(): Retrieves, but does not remove, the element with the highest priority.
3. poll(): Retrieves and removes the element with the highest priority.
4. remove(E e): Removes a specific element from the queue.

Default Behavior (Min-Heap):

By default, a PriorityQueue in Java sorts the elements in **natural order** (ascending). The smallest element has the highest priority.

Example: Min-Heap (Default PriorityQueue)

```java
import java.util.PriorityQueue;

public class MinHeapExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        minHeap.add(10);
        minHeap.add(5);
        minHeap.add(30);
        minHeap.add(15);

        // Poll elements in ascending order (min-heap behavior)
        while (!minHeap.isEmpty()) {
            System.out.println(minHeap.poll());  // Output: 5, 10, 15, 30
        }
    }
}
```

Example: Max-Heap (Custom Comparator)

To turn the PriorityQueue into a max-heap, we need to provide a custom comparator.

```java
import java.util.PriorityQueue;
import java.util.Collections;

public class MaxHeapExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());  // Max-heap

        maxHeap.add(10);
        maxHeap.add(5);
        maxHeap.add(30);
        maxHeap.add(15);

        // Poll elements in descending order (max-heap behavior)
        while (!maxHeap.isEmpty()) {
            System.out.println(maxHeap.poll());  // Output: 30, 15, 10, 5
        }
    }
}
```

How It Works:

- Internally, the PriorityQueue uses a **binary heap** data structure. The heap is a complete binary tree where the parent node is either less than or greater than its children, depending on whether it's a min-heap or max-heap.
- **Min-Heap**: In a min-heap, the smallest element is always at the root of the heap. This ensures that when you poll an element from the priority queue, the smallest element is removed.
- **Max-Heap**: In a max-heap, the largest element is at the root, so the element with the highest value is removed first.

Time Complexity:

- **Adding an element (add)**: O(log n) because the element needs to be placed in the correct position in the heap.
- **Removing the highest/lowest priority element (poll)**: O(log n) for adjusting the heap after removing the root.
- **Accessing the highest/lowest priority element (peek)**: O(1) as the root of the heap is always the highest/lowest element.

Use Cases:

- **Job Scheduling**: Tasks with higher priority (such as real-time tasks) can be executed before others.
- **Dijkstra's Algorithm**: Used to find the shortest path in graph algorithms where a priority queue helps in retrieving the next closest node.
- **Event Simulation**: Events can be processed in the order of their priority (time of occurrence).

Summary:

A PriorityQueue in Java is an efficient way to manage elements where priority is a key concern. It uses a binary heap under the hood to ensure that insertion, removal, and access to the highest-priority element are efficient. By default, it behaves like a min-heap, but with a custom comparator, it can be turned into a max-heap.

❖ Create a PriorityQueue (Max-Heap) to store map entries based on frequency

Here's the completed function to print characters in descending order based on their occurrence frequency in a string. I have made the necessary changes to correctly handle the priority queue and print the results:

```
import java.util.*;

public class CharacterFrequencyDescending {

    public static void main(String[] args) {
        String str = "abbaccaddbb";
        DescendingOrderOfOccurrences(str);
    }

    // Function to print characters in descending order of occurrences
    static void DescendingOrderOfOccurrences(String str) {
        // Step 1: Create a HashMap to store the frequency of each character
        HashMap<Character, Integer> map = new HashMap<>();
        char[] charArr = str.toCharArray();

        // Step 2: Count the occurrences of each character
        for (char ch : charArr) {
            map.put(ch, map.getOrDefault(ch, 0) + 1);
        }

        // Step 3: Create a PriorityQueue (Max-Heap) to store map entries based on frequency
        PriorityQueue<Map.Entry<Character, Integer>> maxHeap = new PriorityQueue<>(
            (a, b) -> b.getValue() - a.getValue()); // Sort by frequency in descending order

        // Step 4: Add all map entries to the maxHeap
        maxHeap.addAll(map.entrySet());

        // Step 5: Poll from the heap and print characters in descending order of frequency
        while (!maxHeap.isEmpty()) {
            Map.Entry<Character, Integer> entry = maxHeap.poll();
            System.out.println("Character: " + entry.getKey() + " Frequency: " + entry.getValue());
        }
    }
}
```

Explanation:

1. **Step 1:** We create a HashMap to count the frequency of each character in the string.
    - map.put(ch, map.getOrDefault(ch, 0) + 1); increments the count of the character in the map.
2. **Step 2:** We convert the map into a PriorityQueue (max-heap) using a custom comparator that sorts entries in descending order based on their values (frequencies).
    - PriorityQueue<Map.Entry<Character, Integer>> maxHeap = new PriorityQueue<>((a, b) -> b.getValue() - a.getValue());: This ensures that entries with the highest frequency come first.
3. **Step 3:** We add all the entries from the map to the PriorityQueue.
4. **Step 4:** We print the characters and their frequencies in descending order by polling from the PriorityQueue.

Example Output:

For the input string "abbaccaddbb", the output will be:

```
Character: b Frequency: 4
Character: a Frequency: 4
Character: c Frequency: 2
Character: d Frequency: 2
```

This function efficiently sorts the characters based on their frequency and prints them in descending order.

❖ Explain this: **PriorityQueue<>((a, b) -> b.getValue() - a.getValue());**

PriorityQueue

A PriorityQueue in Java by default is a **min-heap**, which means it arranges the elements in natural order (ascending). In order to make it a **max-heap** (to get the largest elements first), we need to provide a custom comparator. The comparator dictates how elements are ordered in the queue.

(a, b) -> b.getValue() - a.getValue()

This is a **lambda expression** that defines the custom comparator. It's shorthand for a comparison function used to sort two elements, a and b, in the queue.

- a and b are of type Map.Entry<Character, Integer>, since you're working with entries of a HashMap that stores characters as keys and their frequencies as values.

- a.getValue() and b.getValue() retrieve the frequencies (integer values) of the corresponding characters.

Custom Comparator Logic

The expression (a, b) -> b.getValue() - a.getValue():

- **Subtracts the value of a from b**:
    - If b.getValue() is greater than a.getValue(), this subtraction will yield a positive result, meaning b will come before a, and thus higher frequency elements will be given higher priority in the queue.
    - If a.getValue() is greater than b.getValue(), the result will be negative, meaning a comes before b, giving lower frequency elements lower priority.

1. List Interface

A List is an ordered collection that allows duplicate elements. The list maintains the insertion order.

Implementations of List:

- **ArrayList**
- **LinkedList**
- **Vector**

a) ArrayList

ArrayList in Java is a part of the **Java Collections Framework** and is a **resizable array implementation** of the List interface. It provides the functionality of dynamic arrays, which means the size of an ArrayList can grow or shrink as elements are added or removed.

Key Characteristics of ArrayList:

1. **Dynamic Resizing**: Unlike arrays in Java, which have a fixed size, ArrayList can dynamically resize itself when elements are added or removed.
2. **Index-Based Access**: It allows random access to elements, meaning you can access any element using its index in constant time (O(1)).
3. **Maintains Insertion Order**: Elements are ordered based on their insertion order.
4. **Allows Duplicates**: You can store duplicate elements in an ArrayList.
5. **Non-Synchronized**: ArrayList is not synchronized, meaning it is not thread-safe for concurrent use. You can make it synchronized using Collections.synchronizedList() if needed.

Important Constructors:

- **ArrayList()**: Creates an empty list with an initial capacity of 10.
- **ArrayList(int initialCapacity)**: Creates an empty list with the specified initial capacity.
- **ArrayList(Collection<? extends E> c)**: Creates a list containing the elements of the specified collection.

Example of ArrayList:

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Create an ArrayList of Strings
```

```java
        ArrayList<String> fruits = new ArrayList<>();

        // Add elements to the ArrayList
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");

        // Display the ArrayList
        System.out.println("Fruits: " + fruits);

        // Access elements by index
        System.out.println("First fruit: " + fruits.get(0));

        // Modify an element
        fruits.set(1, "Mango");
        System.out.println("Modified Fruits: " + fruits);

        // Remove an element by index
        fruits.remove(2);
        System.out.println("After removal: " + fruits);

        // Iterate over the ArrayList
        System.out.println("Iterating over ArrayList:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

Output:

```
Fruits: [Apple, Banana, Orange]
First fruit: Apple
Modified Fruits: [Apple, Mango, Orange]
After removal: [Apple, Mango]
Iterating over ArrayList:
Apple
Mango
```

Key Methods of ArrayList:

1. **add(E e)**: Adds an element to the list.
2. **add(int index, E element)**: Adds an element at the specified index.
3. **get(int index)**: Retrieves the element at the specified index.
4. **set(int index, E element)**: Replaces the element at the specified index with the given element.
5. **remove(int index)**: Removes the element at the specified index.
6. **size()**: Returns the number of elements in the list.
7. **clear()**: Removes all elements from the list.
8. **contains(Object o)**: Checks if the list contains the specified element.
9. **isEmpty()**: Returns true if the list is empty, false otherwise.
10. **toArray()**: Converts the ArrayList into an array.

Performance:

- **Access Time**: O(1) – Accessing an element by its index is fast.
- **Insertion/Deletion**: O(n) – Adding or removing elements at arbitrary positions involves shifting elements, which can take linear time.
- **Resizing**: When the array is full, the internal array of the ArrayList is resized by 50% more than its previous size. This can occasionally take extra time, but it happens infrequently.

Example: Adding and Iterating Over Elements:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();

        // Adding elements
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        // Iterate using for-each loop
        for (int number : numbers) {
            System.out.println(number);
        }

        // Accessing a specific element
        System.out.println("Element at index 1: " + numbers.get(1));

        // Removing an element
        numbers.remove(2);

        // Size of ArrayList
        System.out.println("Size: " + numbers.size());
    }
}
```

Conclusion:

ArrayList in Java is a highly versatile and commonly used collection class when you need a dynamic array-like data structure. It is ideal when frequent access by index is required, and the size of the list needs to change dynamically.

- **Internal Implementation**:
  - **Backing structure**: Resizable array (Object[]).
  - **Resizing mechanism**: When an element is added and the array is full, it grows by 50% of its size.
  - **Access time**: Fast random access due to array-based indexing (O(1) for get).
  - **Insertion/Deletion**: Slower (O(n)) since elements must be shifted in case of insertion/deletion at arbitrary positions.
- **Use Cases**:
  - When you need **fast access** to elements via index.

- When the list is frequently **read** but rarely modified.

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
```

b) LinkedList

A **Linked List** is a data structure in which elements (nodes) are stored in a linear sequence, but unlike arrays, these elements are not stored in contiguous memory locations. Instead, each element in a linked list contains two parts:

1. **Data**: The actual value of the element.
2. **Reference (Link/Pointer)**: A pointer or reference to the next element in the sequence.

This means that the elements of the linked list are dynamically allocated, and the order of elements is maintained using the pointers rather than by their physical placement in memory.

Types of Linked Lists:

There are several variations of linked lists, each suited to different use cases:

1. **Singly Linked List**:
   - Each node has one pointer that points to the next node.
   - The last node points to null, indicating the end of the list.

   Structure of a node in a singly linked list:

   [Data | Next]

2. **Doubly Linked List**:
   - Each node has two pointers: one pointing to the next node and one pointing to the previous node.
   - This allows traversal in both directions (forward and backward).

   Structure of a node in a doubly linked list:

   [Prev | Data | Next]

3. **Circular Linked List**:
   - In a circular linked list, the last node points back to the first node instead of pointing to null. This forms a loop.
   - This can be applied to both singly and doubly linked lists.

4. **Circular Doubly Linked List**:
   - Similar to a circular linked list, but with nodes that have both prev and next pointers, forming a bi-directional circular chain.

Basic Operations on Linked Lists:

1. **Traversal**:
   - To visit all the nodes in the linked list.
   - Example (Singly Linked List):

```
Node current = head;
while (current != null) {
  System.out.println(current.data);
```

```
    current = current.next;
}
```

2. **Insertion**:
   - Adding a new node to the linked list.
   - It can be done at the beginning, end, or at any specific position.

Example (Inserting at the beginning in a singly linked list):

```
public void insertAtBeginning(int newData) {
    Node newNode = new Node(newData);
    newNode.next = head;
    head = newNode;
}
```

3. **Deletion**:
   - Removing a node from the linked list.
   - Can be done from the beginning, end, or any specific position.

Example (Deleting a node with a specific key):

```
public void deleteByKey(int key) {
    Node current = head, prev = null;

    // If head node itself holds the key
    if (current != null && current.data == key) {
        head = current.next;  // Change head
        return;
    }

    // Search for the key to be deleted
    while (current != null && current.data != key) {
        prev = current;
        current = current.next;
    }

    // If the key was not present in the list
    if (current == null) return;

    // Unlink the node from the list
    prev.next = current.next;
}
```

4. **Search**:
   - Finding a node with a specific value.
   - Traverse the list and check if the node contains the value.

Example (Search in a singly linked list):

```
public boolean search(int key) {
    Node current = head;
    while (current != null) {
        if (current.data == key)
            return true;
        current = current.next;
    }
    return false;
}
```

Advantages of Linked Lists:

1. **Dynamic Size**: The size of a linked list can grow or shrink dynamically, unlike arrays, which have a fixed size.
2. **Efficient Insertions and Deletions**: Inserting or deleting an element in a linked list is efficient (O(1) time complexity) if we know the position. In contrast, arrays require shifting of elements during insertions and deletions (O(n) time complexity).
3. **Efficient Memory Utilization**: Linked lists do not require contiguous memory allocation, unlike arrays. This helps in utilizing memory efficiently.

Disadvantages of Linked Lists:

1. **Random Access Not Allowed**: Linked lists do not allow direct access to an element like arrays (which use indexes). You have to traverse the list sequentially to reach a particular node, which results in O(n) time complexity for accessing an element.
2. **Extra Memory Overhead**: Each node in a linked list requires extra memory for storing the pointer/reference to the next node, which results in increased memory usage.
3. **Complexity**: Operations like searching for an element or accessing an element by its index take linear time (O(n)) in a linked list compared to constant time (O(1)) in arrays.

Example: Singly Linked List in Java

```
class LinkedList {
    Node head;  // Head of the list

    // Node class
    static class Node {
        int data;
        Node next;

        // Constructor to create a new node
        Node(int d) {
            data = d;
            next = null;
        }
    }

    // Insert a new node at the end
    public void append(int newData) {
        Node newNode = new Node(newData);

        // If the Linked List is empty, make the new node the head
        if (head == null) {
            head = newNode;
            return;
```

```java
        }

        // Else traverse to the end and insert the new node
        Node last = head;
        while (last.next != null) {
            last = last.next;
        }
        last.next = newNode;
    }

    // Print the linked list
    public void printList() {
        Node currNode = head;
        while (currNode != null) {
            System.out.print(currNode.data + " ");
            currNode = currNode.next;
        }
    }

    public static void main(String[] args) {
        LinkedList list = new LinkedList();

        // Adding elements
        list.append(1);
        list.append(2);
        list.append(3);

        // Print the LinkedList
        list.printList();  // Output: 1 2 3
    }
}
```

Example: Doubly Linked List in Java

```java
class DoublyLinkedList {
    Node head;  // Head of the list

    // Node class
    static class Node {
        int data;
        Node prev, next;

        // Constructor to create a new node
        Node(int d) {
            data = d;
            prev = next = null;
        }
    }

    // Insert a new node at the end
    public void append(int newData) {
        Node newNode = new Node(newData);

        // If the list is empty, make the new node the head
        if (head == null) {
            head = newNode;
            return;
        }

        // Else traverse to the end and insert the new node
        Node last = head;
        while (last.next != null) {
            last = last.next;
        }
        last.next = newNode;
        newNode.prev = last;
    }

    // Print the list forward
    public void printListForward() {
        Node currNode = head;
        while (currNode != null) {
            System.out.print(currNode.data + " ");
            currNode = currNode.next;
        }
    }

    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();

        // Adding elements
        list.append(1);
        list.append(2);
        list.append(3);

        // Print the LinkedList
        list.printListForward();  // Output: 1 2 3
    }
}
```

Use Cases of Linked Lists:

- **Dynamic Memory Allocation**: Linked lists are ideal when the size of the list is unknown or frequently changes.
- **Implementation of Stacks, Queues, and Deques**: Linked lists can be used to implement these data structures efficiently.
- **Undo/Redo Functionality**: In editors and other applications, linked lists are useful for implementing undo/redo functionalities.
- **Browser History**: Linked lists can be used to implement browser forward and backward history.

Conclusion:

Linked lists are a versatile and powerful data structure that offers dynamic memory allocation and efficient insertions/deletions. However, they come with the trade-off of slower access times due to the sequential nature of the structure.

- **Internal Implementation**:
    - **Backing structure**: Doubly linked list.

  - **Insertion/Deletion**: Fast insertion and deletion at the beginning or middle (O(1) for add/remove at head).
  - **Access time**: Slow random access (O(n)), since elements must be traversed sequentially.

- **Use Cases**:
  - When the list is frequently **modified** (insertions/deletions).

    When dealing with **frequent modifications** (insertions or deletions), the choice of data structure plays a significant role in performance. Here's why **LinkedList** is a good choice in such scenarios:

1. Frequent Modifications (Insertions/Deletions)

- **LinkedList** is a **doubly linked list**. Each node in a LinkedList contains two references: one to the previous node and one to the next node in the sequence.
- When you insert or delete elements in a LinkedList, there is no need to shift elements like in an array (used in ArrayList). The only operation needed is adjusting the pointers to link or unlink nodes, which happens in constant time **O(1)**.
- **Insertion/Deletion**:
  - **At the beginning or end of the list**: LinkedList excels in these operations because it can easily update the head or tail pointers without needing to shift elements.
  - **At arbitrary positions**: Inserting or deleting elements in the middle of a LinkedList also only requires changing the pointers of the adjacent nodes, making it efficient compared to ArrayList, which would need to shift many elements in memory.

- When you need a **queue** or **stack** behavior since LinkedList implements Deque.

2. Queue or Stack Behavior

- The **LinkedList implements Deque (Double Ended Queue)**, which means it can act both as a **queue** and a **stack**. It allows insertion and deletion of elements from both ends of the list.
- **Queue behavior** (First-In-First-Out - FIFO):
  - A **queue** processes elements in the order they were added. The addLast() method adds elements to the end, while the removeFirst() removes elements from the front.
  - In LinkedList, this is efficiently handled since adding or removing elements from either end of the list takes constant time, **O(1)**.

```
Deque<String> queue = new LinkedList<>();
queue.addLast("Apple");  // Add to the end of the queue
queue.addLast("Banana");
String first = queue.removeFirst(); // Remove from the front (FIFO)
```

- **Stack behavior** (Last-In-First-Out - LIFO):
  - A **stack** processes elements in reverse order of how they were added (last in, first out). The addFirst() method pushes elements to the front, and removeFirst() pops them from the front.
  - Again, LinkedList allows constant-time **O(1)** operations for pushing and popping from either end.

```
Deque<String> stack = new LinkedList<>();
stack.addFirst("Apple");  // Push to the front (stack behavior)
stack.addFirst("Banana");
String last = stack.removeFirst(); // Pop from the front (LIFO)
```

Why LinkedList is Efficient for these Behaviors

- **Queue behavior**: You often insert at the end and remove from the front. LinkedList allows both these operations to happen in constant time.
- **Stack behavior**: You insert and remove from the same end. LinkedList can easily handle this with addFirst() and removeFirst() methods, making it a perfect fit for stack-like usage.

For example:

```
LinkedList<String> list = new LinkedList<>();
list.add("Apple"); // Adding element at the end (O(1))
list.addFirst("Orange");  // Adding element at the beginning (O(1))
list.remove(1);    // Removing element at index 1 (O(1))
```

```
List<String> list = new LinkedList<>();
list.add("Apple");
list.add("Banana");
```

c) Vector

A **Vector** is a class in the java.util package that implements a dynamic array, which means it can grow or shrink as needed to accommodate elements. It is part of the legacy collection framework but has been retrofitted to implement the List interface, so it is similar to an ArrayList. However, unlike ArrayList, **Vector** is synchronized, meaning it is thread-safe, making it suitable for multi-threaded environments.

Key Features of Vector in Java:

1. **Dynamic Size**: Unlike arrays, the size of a Vector can grow or shrink as elements are added or removed.
2. **Synchronized**: Vector is thread-safe because all its methods are synchronized. This means only one thread can access the vector at a time, ensuring consistency in multi-threaded environments.
3. **Random Access**: Like arrays and ArrayList, Vector allows random access to elements through an index, which makes it faster for element retrieval.
4. **Duplicates Allowed**: It allows duplicate elements.
5. **Maintains Insertion Order**: Elements in a Vector are stored in the order in which they were added.

Vector Constructors:

1. Vector(): Creates a default vector with an initial capacity of 10.
2. Vector(int capacity): Creates a vector with the specified initial capacity.
3. Vector(int capacity, int increment): Creates a vector with the specified initial capacity and a capacity increment, meaning the vector's capacity increases by a specified amount when needed.
4. Vector(Collection<? extends E> c): Creates a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Common Methods of Vector:

Here are some commonly used methods in the Vector class:

1. **add(E e)**: Appends the specified element to the end of this vector.

2. **add(int index, E element)**: Inserts the specified element at the specified position in the vector.
3. **remove(int index)**: Removes the element at the specified position in the vector.
4. **remove(Object o)**: Removes the first occurrence of the specified element from the vector, if it is present.
5. **get(int index)**: Returns the element at the specified position in the vector.
6. **set(int index, E element)**: Replaces the element at the specified position in the vector with the specified element.
7. **size()**: Returns the number of elements in the vector.
8. **isEmpty()**: Returns true if the vector contains no elements.
9. **clear()**: Removes all elements from the vector.
10. **contains(Object o)**: Returns true if the vector contains the specified element.
11. **capacity()**: Returns the current capacity of the vector.
12. **firstElement()**: Returns the first element of the vector.
13. **lastElement()**: Returns the last element of the vector.
14. **elementAt(int index)**: Returns the element at the specified index.
15. **indexOf(Object o)**: Returns the index of the first occurrence of the specified element in the vector.

Example: Basic Operations with Vector

```
import java.util.Vector;

public class VectorExample {
    public static void main(String[] args) {
        // Creating a vector with initial capacity 5
        Vector<Integer> vector = new Vector<>(5);

        // Adding elements
        vector.add(10);
        vector.add(20);
        vector.add(30);
        vector.add(40);

        // Inserting an element at a specific position
        vector.add(1, 15);

        // Accessing elements
        System.out.println("Element at index 2: " + vector.get(2));  // Output: 20

        // Removing an element
        vector.remove(3);  // Removes the element at index 3 (i.e., 30)

        // Size and capacity of the vector
        System.out.println("Size of the vector: " + vector.size());  // Output: 4
        System.out.println("Capacity of the vector: " + vector.capacity());  // Output: 5

        // Checking if a particular element exists
        System.out.println("Does vector contain 20? " + vector.contains(20));  // Output: true

        // Iterating over the vector
        System.out.println("Elements in the vector:");
        for (Integer element : vector) {
            System.out.println(element);
        }
    }
}
```

Output:

```
Element at index 2: 20
Size of the vector: 4
Capacity of the vector: 5
Does vector contain 20? true
Elements in the vector:
10
15
20
40
```

When to Use Vector:

- **Thread-Safety**: If your application requires synchronization and thread-safety while working with dynamic arrays, Vector is a good option.
- **Multi-threaded Environments**: In cases where multiple threads need to access and modify the same collection, Vector ensures that only one thread can perform these operations at a time.

When Not to Use Vector:

- **Performance Concerns**: Due to the synchronization, Vector can be slower than non-synchronized alternatives like ArrayList in a single-threaded environment.
- **Modern Java**: In modern Java development, Vector has largely been replaced by ArrayList or other non-synchronized collections. If thread-safety is required, it is more common to use ArrayList along with external synchronization (or alternatives like CopyOnWriteArrayList).

Conclusion:

The Vector class provides a dynamic array that can grow as needed and ensures thread-safety by synchronizing its methods. However, its usage has declined in favor of ArrayList, which provides better performance in single-threaded applications.

- **Internal Implementation**:
  - Similar to ArrayList, but it is **synchronized**. Uses a resizable array.
  - **Thread-safe**, but slower due to synchronization overhead.
  - **Resizing mechanism**: Grows by 100% when full.
- **Use Cases**:
  - When you need a thread-safe alternative to ArrayList but don't want to manage synchronization manually.

The Vector class is a part of the java.util package. It is similar to an ArrayList, but with two main differences:

1. **Thread Safety**: Vector methods are synchronized, making it thread-safe.
2. **Legacy Class**: It is a legacy class, meaning it has been part of Java since version 1.0.

Features:

- Like ArrayList, Vector is a dynamically resizable array, meaning it can grow and shrink as needed.

- Vector maintains insertion order.
- Vector can hold duplicates and allows null elements.

Example of Using Vector in Java:

```
import java.util.Vector;
public class VectorExample {
    public static void main(String[] args) {
        // Create a Vector
        Vector<String> vector = new Vector<>();

        // Add elements to the Vector
        vector.add("Apple");
        vector.add("Banana");
        vector.add("Orange");

        // Display elements in the Vector
        System.out.println("Vector Elements: " + vector);

        // Add an element at a specific index
        vector.add(1, "Mango");
        System.out.println("After inserting Mango at index 1: " + vector);

        // Remove an element from the Vector
        vector.remove("Banana");
        System.out.println("After removing Banana: " + vector);

        // Get an element at a specific index
        String fruit = vector.get(2);
        System.out.println("Element at index 2: " + fruit);

        // Iterating through the Vector using a for-each loop
        System.out.println("Iterating over Vector:");
        for (String item : vector) {
            System.out.println(item);
        }
    }
}
```

Output:

```
Vector Elements: [Apple, Banana, Orange]
After inserting Mango at index 1: [Apple, Mango, Banana, Orange]
After removing Banana: [Apple, Mango, Orange]
Element at index 2: Orange
Iterating over Vector:
Apple
Mango
Orange
```

Key Methods:

- add(E e): Appends the specified element to the end of this Vector.
- add(int index, E element): Inserts the specified element at the specified position in this Vector.
- remove(Object o): Removes the first occurrence of the specified element from this Vector.
- get(int index): Returns the element at the specified position in this Vector.

- size(): Returns the number of elements in the Vector.

Though Vector provides thread-safety, in most cases, it's recommended to use ArrayList or other concurrent collections unless thread-safety is explicitly required.

---

2. Set Interface

A Set is a collection that **does not allow duplicates** and may or may not maintain the order of elements.

Implementations of Set:

- **HashSet**
- **LinkedHashSet**
- **TreeSet**

a) HashSet

In Java, HashSet is a part of the java.util package and implements the Set interface. It is backed by a **hash table** (actually a HashMap), and it **does not allow duplicate elements**. It provides **constant time performance** for basic operations like add, remove, and contains, assuming the hash function disperses elements properly among the buckets.

Key Characteristics of HashSet:

1. **No Duplicates**: A HashSet cannot contain duplicate elements.
2. **Unordered**: The elements in a HashSet are not ordered. The iteration order of elements is not guaranteed and can change over time.
3. **Null Elements**: It allows a single null element.
4. **Thread-Safety**: HashSet is **not synchronized**. If multiple threads access a HashSet concurrently, it must be synchronized externally.

Common Methods in HashSet:

- add(E e): Adds the specified element to the set if it is not already present.
- remove(Object o): Removes the specified element from the set if it is present.
- contains(Object o): Returns true if the set contains the specified element.
- size(): Returns the number of elements in the set.
- isEmpty(): Returns true if the set contains no elements.

Example of HashSet in Java:

```
import java.util.HashSet;
import java.util.Set;

public class HashSetExample {
    public static void main(String[] args) {
        // Create a HashSet of Strings
        Set<String> hashSet = new HashSet<>();

        // Add elements to the set
        hashSet.add("Apple");
        hashSet.add("Banana");
        hashSet.add("Orange");
        hashSet.add("Apple"); // This will not be added as it's a duplicate

        // Display elements in the set
        System.out.println("HashSet Elements: " + hashSet);

        // Check if an element exists
        boolean hasBanana = hashSet.contains("Banana");
        System.out.println("Does the set contain Banana? " + hasBanana);

        // Remove an element
        hashSet.remove("Orange");
        System.out.println("After removing Orange: " + hashSet);

        // Get the size of the set
        System.out.println("Size of the set: " + hashSet.size());

        // Iterate over the elements in the set
        System.out.println("Iterating over HashSet:");
        for (String fruit : hashSet) {
            System.out.println(fruit);
        }
    }
}
```

Output:

```
HashSet Elements: [Apple, Orange, Banana]  // Order may vary
Does the set contain Banana? true
After removing Orange: [Apple, Banana]
Size of the set: 2
Iterating over HashSet:
Apple
Banana
```

Internal Working of HashSet:

1. **Hashing**: When an object is added to a HashSet, Java uses the object's hashCode() method to calculate its hash code. This hash code is used to determine the bucket where the object will be stored.
2. **Handling Collisions**: If two objects have the same hash code (hash collision), they are stored in a linked list at the same bucket.
3. **Equality Check**: HashSet uses the equals() method to check whether two objects are identical or not. Even if two objects have the same hash code, the equals() method ensures that no duplicates are added.

Performance of HashSet:

- The time complexity for the basic operations (add, remove, contains) is **O(1)** in the best case (constant time) but can degrade to **O(n)** in the worst case (if many elements have the same hash code, causing collisions).

Key Points:

- Use HashSet when you do not care about the order of elements and you want to prevent duplicates.
- If you need a **sorted set**, use TreeSet.
- If you need to maintain **insertion order**, use LinkedHashSet.

Synchronized HashSet:

If you need a thread-safe HashSet, you can wrap it using Collections.synchronizedSet():

```
Set<String> synchronizedSet = Collections.synchronizedSet(new HashSet<>());
```

- **Internal Implementation**:
    - **Backing structure**: Hash table (based on HashMap).
    - **Time complexity**: Constant time (O(1)) for add, remove, and contains, assuming good hash distribution.
    - **Order**: Unordered, does not maintain insertion order.
- **Use Cases**:
    - When you need to store unique elements, and **order does not matter**.
    - Best suited for large datasets where you frequently check for **membership**.

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
```

Internal working of HashSet

The internal working of HashSet in Java revolves around concepts like hash tables, hash codes, and collision handling. To fully understand this, we need to break down each of these concepts in more detail:

1. Hash Table

A **hash table** is a data structure that stores key-value pairs. It provides efficient access, insertion, and deletion operations, typically in constant time (O(1)), based on the hash code of the key.

- **Structure**: A hash table consists of an array of **buckets**, where each bucket can store one or more elements. Each element in a hash table is placed into a bucket based on its hash code.
- **Buckets**: A bucket is essentially a container (array or linked list) inside the hash table. If two or more elements end up in the same bucket due to a hash code collision (discussed later), the bucket will store multiple elements.

In HashSet, each element is stored as the key in an internal HashMap, and the value is a constant PRESENT (just a dummy value, since HashSet only cares about keys).

2. Hash Code

A **hash code** is an integer value returned by the hashCode() method of an object in Java. Every object in Java has a hashCode() method (inherited from Object) that returns an integer representing the object. This hash code is used to determine the index (bucket) in the hash table where the object should be placed.

- **Purpose of hashCode()**: The hashCode() method helps quickly locate an element in a hash table. Instead of checking each element one by one, the hash code allows the program to calculate which bucket an element belongs to.

For example:

```
public int hashCode() {
   return 31 * value1 + value2; // Generates an integer based on the object's state
}
```

- **Hash Code Range**: The hashCode() method returns an int which could be positive or negative. The size of the array (bucket array) used in the hash table is usually smaller than the possible range of hash codes, so the hash code is used with **modulus operation** to fit into the array size.

For example, if the size of the bucket array is 16, the hash code could be used as:

int index = hashCode() % 16; // This gives the bucket index for the element

3. Bucket

A **bucket** is essentially a storage location in the hash table where elements are placed based on their hash code. When an element is added to the hash table, the hash code is computed and used to find the appropriate bucket. Each bucket can hold multiple elements if necessary (in case of collisions).

- **Example**: Let's say we have a hash table with 10 buckets (indexed from 0 to 9). If we want to add an element and its hash code is 35, the index (bucket) will be determined by calculating 35 % 10, which equals 5. So, the element is stored in the bucket at index 5.

4. Hashing

Hashing is the process of converting an object into a numerical value (the hash code) using a hash function. In Java, the hash code is computed by the hashCode() method.

- **Why Hashing?**: Hashing is used to transform the data (in this case, the object) into a numerical index that can be used to quickly store and retrieve the object in the hash table. This makes it very efficient to locate elements in a large set of data.

- **Hash Function**: The hash function used by Java (inside hashCode() method) transforms the internal state of the object (its fields) into an integer, which is the hash code.

5. Collision Handling

A **collision** occurs when two different elements have the same hash code or when two hash codes, when used with the modulus operation, result in the same bucket index.

For example:

- hashCode("Apple") % 10 might equal 5.

- hashCode("Orange") % 10 might also equal 5.

This means that both "Apple" and "Orange" will be placed in the same bucket. To handle collisions, Java uses different strategies, and the most common one in HashSet is **separate chaining**.

Separate Chaining:

In separate chaining, each bucket in the hash table contains a **linked list** (or another collection), and when multiple elements map to the same bucket, they are stored in that bucket's linked list.

- **Adding Elements**: When adding a new element, the program will compute the hash code to find the correct bucket. If the bucket already contains elements (i.e., a collision has occurred), the new element is added to the linked list at that bucket.

- **Searching Elements**: When searching for an element, the hash code is used to find the appropriate bucket. If the bucket contains multiple elements (due to a collision), the linked list is traversed and equals() is used to compare the elements.

- **Example**: Let's say we add "Apple" and "Orange" to the same bucket due to a collision. The bucket will contain a linked list with both elements. When we later search for "Orange", the program will first find the bucket based on the hash code, then check each element in the linked list using equals() until it finds "Orange".

6. Ensuring Uniqueness (equals() and hashCode() relationship)

One of the key features of a HashSet is that it doesn't allow duplicate elements. To ensure that duplicates are not added, HashSet uses both the hashCode() and equals() methods.

- **hashCode()**: First, the hash code of the element is calculated. This helps narrow down the search to a specific bucket.

- **equals()**: Once the bucket is found, HashSet uses the equals() method to check if the element is already present in that bucket. If an element with the same hash code and equals() result is found, the new element is not added (since it is considered a duplicate).
- **Why Both hashCode() and equals()?**:
  - hashCode() is used for efficient lookup (i.e., to quickly locate the bucket).
  - equals() ensures that even if two objects have the same hash code (a collision), they are only considered equal if their actual content is the same.

For example:

- hashCode("Apple") might give 12345.
- hashCode("Banana") might also give 12345 (this is called a collision).
- However, when equals() is called, it will correctly determine that "Apple" and "Banana" are not equal, so both will be stored in the hash table.

Example of HashSet:

Let's look at an example:

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");
        set.add("Apple");  // Duplicate, will not be added

        System.out.println(set);
    }
}
```

Step-by-Step Breakdown:

1. When "Apple" is added, its hashCode() is computed. The hash code is used to determine the bucket where "Apple" will be stored.
2. When "Banana" is added, its hashCode() is computed, and its bucket is found.
3. When "Apple" is added again, its hash code is computed, and the equals() method is used to check if it already exists in the set (it does), so it is not added again.

Conclusion:

The internal working of HashSet is centered around efficient element management using hash codes and hash tables. By leveraging hashing, collisions, and the combination of hashCode() and equals(), HashSet ensures quick access to elements and enforces uniqueness.

b) LinkedHashSet

In Java, LinkedHashSet is a class that implements the Set interface and extends HashSet. It maintains a **linked list** of the set elements, which means it preserves the **insertion order**. The LinkedHashSet provides all the functionalities of a HashSet, but with the added benefit of remembering the order in which elements were inserted.

Key Characteristics of LinkedHashSet:

1. **Preserves Insertion Order**: Unlike HashSet, LinkedHashSet maintains the order of elements as they are inserted.
2. **No Duplicates**: Like all sets, it does not allow duplicate elements.
3. **Performance**: The performance of LinkedHashSet is slightly slower than HashSet due to the added overhead of maintaining the linked list, but it still provides constant time (O(1)) performance for basic operations like add, remove, and contains.
4. **Null Elements**: It allows one null element, just like HashSet.

Common Methods in LinkedHashSet:

- add(E e): Adds the specified element to the set if it is not already present.
- remove(Object o): Removes the specified element from the set if it is present.
- contains(Object o): Returns true if the set contains the specified element.
- size(): Returns the number of elements in the set.
- isEmpty(): Returns true if the set contains no elements.
- iterator(): Returns an iterator to traverse the elements in insertion order.

Example of LinkedHashSet in Java:

```
import java.util.LinkedHashSet;
import java.util.Set;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        // Create a LinkedHashSet of Strings
        Set<String> linkedHashSet = new LinkedHashSet<>();

        // Add elements to the set
        linkedHashSet.add("Apple");
        linkedHashSet.add("Banana");
        linkedHashSet.add("Orange");
        linkedHashSet.add("Apple"); // Duplicate, will not be added

        // Display elements in the set (insertion order is maintained)
        System.out.println("LinkedHashSet Elements: " + linkedHashSet);

        // Check if an element exists
        boolean hasBanana = linkedHashSet.contains("Banana");
        System.out.println("Does the set contain Banana? " + hasBanana);

        // Remove an element
        linkedHashSet.remove("Orange");
        System.out.println("After removing Orange: " + linkedHashSet);

        // Get the size of the set
        System.out.println("Size of the set: " + linkedHashSet.size());
        // Iterate over the elements in insertion order
        System.out.println("Iterating over LinkedHashSet:");
        for (String fruit : linkedHashSet) {
            System.out.println(fruit);
        }
    }
}
```

Output:

```
LinkedHashSet Elements: [Apple, Banana, Orange]  // Insertion order preserved
Does the set contain Banana? true
After removing Orange: [Apple, Banana]
Size of the set: 2
Iterating over LinkedHashSet:
Apple
Banana
```

Internal Working of LinkedHashSet:

- **Hashing**: Like HashSet, LinkedHashSet is backed by a hash table. It uses the hashCode() method to determine where to place elements in the hash table.
- **Linked List**: In addition to hashing, LinkedHashSet maintains a doubly linked list that records the insertion order of elements. This is why the order is preserved when you iterate through the set.

Performance of LinkedHashSet:

- **Time Complexity**: For basic operations like add, remove, and contains, the time complexity is typically **O(1)**, though it might be slightly slower than HashSet due to the overhead of maintaining the linked list.
- **Memory Overhead**: LinkedHashSet requires more memory than HashSet because of the linked list structure used to maintain the order of elements.

When to Use LinkedHashSet:

- Use LinkedHashSet when you need a Set that:
  1. Does not allow duplicates.
  2. Preserves the order in which elements were inserted.

For example, if you're working with a collection of elements where the order of insertion is important, like maintaining a history of unique user actions or ordered unique data, LinkedHashSet is a good choice.

Comparison with Other Set Implementations:

1. **HashSet**: Does not preserve any order.
2. **LinkedHashSet**: Preserves insertion order.
3. **TreeSet**: Sorts elements based on their natural order or a custom comparator, but it does not preserve insertion order.

Synchronized LinkedHashSet:

Like HashSet, LinkedHashSet is **not synchronized**. If you need to use it in a multi-threaded environment, you can wrap it using Collections.synchronizedSet():

```
Set<String> synchronizedSet = Collections.synchronizedSet(new LinkedHashSet<>());
```

Conclusion:

The LinkedHashSet class is useful when you want a set that maintains insertion order while ensuring that duplicates are not allowed. It provides a good balance between HashSet's performance and TreeSet's sorting but with insertion-order retention.

- **Internal Implementation**:
  - Uses a combination of a hash table and a linked list.
  - **Maintains insertion order**.
  - Slightly slower than HashSet but keeps order intact.
- **Use Cases**:
  - When you need to store unique elements and want to **maintain insertion order**.

```
Set<String> set = new LinkedHashSet<>();
set.add("Apple");
set.add("Banana");
```

c) TreeSet

In Java, TreeSet is a part of the java.util package and implements the Set interface. It is based on a **TreeMap** and is used to store elements in a **sorted order**. Unlike HashSet and LinkedHashSet, which do not guarantee order, a TreeSet stores elements in their **natural order** (or according to a custom comparator, if provided).

Key Characteristics of TreeSet:

1. **Sorted Order**: Elements in a TreeSet are sorted in **ascending** (natural) order by default. You can also define a custom sorting order using a comparator.
2. **No Duplicates**: Like all sets, TreeSet does not allow duplicate elements.
3. **Null Elements**: Before Java 7, TreeSet allowed one null element, but in Java 8 and later, inserting a null value throws a NullPointerException.
4. **Performance**: TreeSet provides log(n) time complexity for basic operations like add, remove, and contains since it is implemented using a Red-Black Tree (a balanced binary search tree).

Common Methods in TreeSet:

- add(E e): Adds the specified element to the set if it is not already present.
- remove(Object o): Removes the specified element from the set if it is present.
- contains(Object o): Returns true if the set contains the specified element.
- size(): Returns the number of elements in the set.
- isEmpty(): Returns true if the set contains no elements.
- first(): Returns the first (lowest) element in the set.
- last(): Returns the last (highest) element in the set.
- pollFirst(): Retrieves and removes the first (lowest) element in the set.
- pollLast(): Retrieves and removes the last (highest) element in the set.

- headSet(E toElement): Returns a view of the portion of the set whose elements are strictly less than toElement.
- tailSet(E fromElement): Returns a view of the portion of the set whose elements are greater than or equal to fromElement.

Example of TreeSet in Java:

```java
import java.util.TreeSet;
public class TreeSetExample {
    public static void main(String[] args) {
        // Create a TreeSet of Integers
        TreeSet<Integer> treeSet = new TreeSet<>();

        // Add elements to the set
        treeSet.add(10);
        treeSet.add(5);
        treeSet.add(20);
        treeSet.add(15);
        treeSet.add(5);  // Duplicate, will not be added

        // Display elements in sorted order
        System.out.println("TreeSet Elements (Sorted): " + treeSet);

        // Get the first (lowest) element
        System.out.println("First Element: " + treeSet.first());

        // Get the last (highest) element
        System.out.println("Last Element: " + treeSet.last());

        // Remove an element
        treeSet.remove(15);
        System.out.println("After removing 15: " + treeSet);

        // Get a subset (headSet)
        System.out.println("Elements less than 15: " + treeSet.headSet(15));

        // Iterate over the elements
        System.out.println("Iterating over TreeSet:");
        for (Integer number : treeSet) {
            System.out.println(number);
        }
    }
}
```

Output:

```
TreeSet Elements (Sorted): [5, 10, 15, 20]
First Element: 5
Last Element: 20
After removing 15: [5, 10, 20]
Elements less than 15: [5, 10]
Iterating over TreeSet:
5
10
20
```

Custom Sorting with TreeSet:

You can define a custom sorting order by passing a Comparator when creating the TreeSet. For example, if you want to store numbers in descending order:

```java
import java.util.Comparator;
import java.util.TreeSet;

public class TreeSetCustomSorting {
    public static void main(String[] args) {
        // Create a TreeSet with a custom comparator (for reverse order)
        TreeSet<Integer> treeSet = new TreeSet<>(Comparator.reverseOrder());

        // Add elements to the set
        treeSet.add(10);
        treeSet.add(5);
        treeSet.add(20);
        treeSet.add(15);

        // Display elements in custom sorted order (descending order)
        System.out.println("TreeSet Elements (Descending): " + treeSet);
    }
}
```

Output:

```
TreeSet Elements (Descending): [20, 15, 10, 5]
```

Internal Working of TreeSet:

- TreeSet is implemented using a **Red-Black Tree**, a type of self-balancing binary search tree.
- The elements are stored according to their natural ordering, or according to the ordering defined by a provided Comparator.

Performance of TreeSet:

- **Time Complexity**: Basic operations like add, remove, and contains have **O(log n)** time complexity because TreeSet is implemented using a balanced tree (Red-Black Tree).

  Limitations:

- TreeSet does not allow null elements in Java 8 and later.
- TreeSet provides slightly slower performance compared to HashSet or LinkedHashSet because it needs to maintain sorting.

When to Use TreeSet:

- You should use TreeSet when you need:
  1. A collection of elements where duplicates are not allowed.
  2. Elements to be stored in sorted (natural or custom) order.
  3. A sorted data structure where frequent retrieval of elements in a particular order is required.

Comparison with Other Set Implementations:

1. **HashSet**: No order is maintained; provides O(1) time complexity for operations.

2. **LinkedHashSet**: Maintains insertion order; provides O(1) time complexity for operations.

3. **TreeSet**: Maintains sorted order; provides O(log n) time complexity for operations.

   Synchronized TreeSet:

   TreeSet is **not synchronized**, so if you need to use it in a multi-threaded environment, you can use Collections.synchronizedSortedSet():

```
SortedSet<Integer> synchronizedTreeSet = Collections.synchronizedSortedSet(new TreeSet<>());
```

Conclusion:

TreeSet is ideal when you need to store elements in a **sorted order** and don't require duplicates. It provides good performance for retrieval operations in sorted order but is slower than HashSet or LinkedHashSet due to the overhead of maintaining the sorting order.

- **Internal Implementation**:
  - **Backing structure**: Red-Black Tree.
  - **Time complexity**: Logarithmic time (O(log n)) for add, remove, and contains.
  - **Order**: Sorted according to natural ordering or a specified comparator.
- **Use Cases**:
  - When you need a sorted set of unique elements.

```
Set<String> set = new TreeSet<>();
set.add("Apple");
set.add("Banana");
```

Explain how set work internally

In Java, the **Set** interface is part of the **Java Collections Framework** and represents a collection of unique elements. Sets do not allow duplicate elements. The common implementations of the Set interface are HashSet, LinkedHashSet, and TreeSet. Each of these has its own internal mechanism for how it stores and manages elements.

To understand how a Set works internally, it's essential to look at how these specific implementations work.

1. HashSet

HashSet is the most commonly used implementation of the Set interface. It stores elements in a **hash table**, which means it uses **hashing** to determine where to place the elements internally.

Internal Working of HashSet:

- **Hash Table**: Internally, HashSet uses a HashMap to store its elements. Each element you add to a HashSet is stored as a key in a HashMap, and the value is a constant (PRESENT) which is a placeholder.
- **Hashing**: When an element is added to the set, its hash code is computed using the hashCode() method of the object. This hash code is then used to determine the bucket (index) in which the element will be placed in the hash table.
- **Collision Handling**: Since two elements can have the same hash code, HashSet uses **separate chaining** to resolve collisions. This means that each bucket in the hash table contains a linked list, and elements with the same hash code are stored in the same bucket but in a linked list.

- **Unique Elements**: Before adding an element to the set, the equals() method is used to check if the element is already present. If an element with the same hash code and equals result exists, the new element is not added (since sets do not allow duplicates).

Example:

```
Set<String> hashSet = new HashSet<>();
hashSet.add("Apple");
hashSet.add("Banana");
hashSet.add("Cherry");
```

Performance:

- HashSet provides constant time (O(1)) performance for basic operations like add(), remove(), and contains(), assuming the hash function distributes the elements well.

2. LinkedHashSet

LinkedHashSet is a subclass of HashSet that maintains the insertion order of the elements. It uses a **doubly linked list** in addition to the hash table to keep track of the order in which elements are inserted.

Internal Working of LinkedHashSet:

- **Hash Table + Linked List**: Like HashSet, it uses a hash table to store elements, but it also uses a doubly linked list to maintain the order of elements.
- **Insertion Order**: When an element is added, the linked list ensures that the element is linked to the previous and next elements, preserving the insertion order.

Example:

```
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Apple");
linkedHashSet.add("Banana");
linkedHashSet.add("Cherry");
```

Performance:

- LinkedHashSet has slightly worse performance than HashSet because it maintains the insertion order, but it still provides constant time (O(1)) for basic operations like add(), remove(), and contains().

3. TreeSet

TreeSet is a part of the java.util package and implements the Set interface using a **red-black tree**, which is a self-balancing binary search tree. This means it sorts the elements in **natural order** or according to a custom Comparator provided.

Internal Working of TreeSet:

- **Red-Black Tree**: TreeSet uses a red-black tree, which ensures that the tree remains balanced. In a balanced tree, the operations to add, remove, or search elements have a logarithmic time complexity.
- **Sorting**: The elements in a TreeSet are sorted according to their natural order (as defined by the Comparable interface), or you can provide a Comparator for custom sorting.
- **No Duplicates**: Like other sets, TreeSet ensures that no duplicate elements are added by checking the equality of elements through comparison.

Example:

```
Set<String> treeSet = new TreeSet<>();
treeSet.add("Banana");
treeSet.add("Apple");
treeSet.add("Cherry");
```

Performance:

- TreeSet provides logarithmic time (O(log n)) performance for operations like add(), remove(), and contains() due to the tree structure.

Comparison of Set Implementations:

Feature	**HashSet**	**LinkedHashSet**	**TreeSet**
Ordering	Unordered	Maintains insertion order	Sorted (natural/custom)
Internal Structure	Hash Table	Hash Table + Linked List	Red-Black Tree
Performance	O(1) for add, remove, search	O(1) but slower than HashSet due to ordering	O(log n) for add, remove, search
Thread-safe	No	No	No
Duplicates	Not allowed	Not allowed	Not allowed

Example of How HashSet Works Internally:

Let's consider an example where we add elements to a HashSet:

```
HashSet<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Cherry");
set.add("Apple"); // Duplicate element, will not be added
```

Step-by-Step Internal Working of HashSet:

1. **"Apple"**:
   - The hashCode() of "Apple" is calculated, and the corresponding bucket is found in the hash table.
   - Since the bucket is empty, "Apple" is added to the hash table.
2. **"Banana"**:
   - The hashCode() of "Banana" is calculated, and its bucket is determined.
   - "Banana" is added to its respective bucket in the hash table.
3. **"Cherry"**:
   - The hashCode() of "Cherry" is computed, and the bucket is found.
   - "Cherry" is added to the hash table.
4. **"Apple" (Duplicate)**:
   - The hashCode() of "Apple" is computed again, and the bucket is checked.
   - The equals() method is used to compare "Apple" with the existing elements in the bucket.
   - Since "Apple" is already present in the set (checked using equals()), it is not added again.

Conclusion:

In summary, the Set interface ensures that elements are unique, and different implementations (HashSet, LinkedHashSet, TreeSet) provide various ways to store and organize these elements. HashSet uses a hash table, LinkedHashSet maintains insertion order, and TreeSet provides a sorted set using a red-black tree.

3. Map Interface

A Map is a collection of **key-value pairs** where keys are unique. The map interface does not extend Collection.

Implementations of Map:

- **HashMap**
- **LinkedHashMap**
- **TreeMap**
- **ConcurrentHashMap**

a) HashMap

In Java, HashMap is part of the java.util package and is one of the most widely used **Map** implementations. It stores data in **key-value pairs** and uses a **hashing** technique to efficiently store and retrieve elements. HashMap allows fast access to elements based on their keys and provides constant-time complexity (O(1)) for basic operations like get and put in the best case.

Key Characteristics of HashMap:

1. **No Duplicate Keys**: HashMap does not allow duplicate keys. Each key must be unique, but multiple keys can have the same value.
2. **Allows Null Keys and Values**: HashMap allows one null key and multiple null values.
3. **Unordered**: HashMap does not guarantee any specific order for the elements (no insertion order or sorted order).
4. **Performance**: HashMap provides constant-time performance for basic operations (put, get, remove) in the best case.
5. **Non-Synchronized**: HashMap is not synchronized, meaning it is not thread-safe for concurrent access. If used in a multithreaded environment, it needs to be synchronized manually or use ConcurrentHashMap.

Common Methods in HashMap:

- put(K key, V value): Adds the specified key-value pair to the map. If the key already exists, the value is updated.
- get(Object key): Returns the value associated with the specified key, or null if the key is not found.
- remove(Object key): Removes the entry with the specified key from the map.
- containsKey(Object key): Returns true if the map contains the specified key.
- containsValue(Object value): Returns true if the map contains one or more keys mapping to the specified value.
- size(): Returns the number of key-value mappings in the map.
- isEmpty(): Returns true if the map contains no key-value mappings.
- keySet(): Returns a Set view of the keys contained in the map.
- values(): Returns a Collection view of the values contained in the map.
- entrySet(): Returns a Set view of the mappings (key-value pairs) contained in the map.

Example of HashMap in Java:

```java
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        // Create a HashMap
        HashMap<String, Integer> hashMap = new HashMap<>();

        // Add key-value pairs to the map
        hashMap.put("Apple", 10);
        hashMap.put("Banana", 20);
        hashMap.put("Orange", 30);
        hashMap.put("Mango", 40);

        // Display the HashMap
        System.out.println("HashMap: " + hashMap);

        // Get a value from the map
        int value = hashMap.get("Banana");
        System.out.println("Value for key 'Banana': " + value);

        // Remove a key-value pair
        hashMap.remove("Mango");
        System.out.println("After removing 'Mango': " + hashMap);

        // Check if a key exists
        boolean hasApple = hashMap.containsKey("Apple");
        System.out.println("Does the map contain 'Apple'? " + hasApple);

        // Iterate over the keys
        System.out.println("Keys in HashMap:");
        for (String key : hashMap.keySet()) {
            System.out.println(key);
        }

        // Iterate over the values
        System.out.println("Values in HashMap:");
        for (Integer val : hashMap.values()) {
            System.out.println(val);
        }

        // Iterate over the key-value pairs (entries)
        System.out.println("Key-Value pairs in HashMap:");
        for (Map.Entry<String, Integer> entry : hashMap.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Output:

```
HashMap: {Apple=10, Banana=20, Orange=30, Mango=40}
Value for key 'Banana': 20
After removing 'Mango': {Apple=10, Banana=20, Orange=30}
Does the map contain 'Apple'? true
Keys in HashMap:
Apple
Banana
```

```
Orange
Values in HashMap:
10
20
30
Key-Value pairs in HashMap:
Apple -> 10
Banana -> 20
Orange -> 30
```

Internal Working of HashMap:

HashMap internally uses an array of buckets (usually called a **hash table**) and a **hashing function** to determine the index of each key-value pair in the array. When two keys produce the same hash value (i.e., a **collision** occurs), HashMap stores these entries in a linked list (or a balanced tree for high collisions) at the corresponding bucket.

Performance of HashMap:

- **Time Complexity**: In the average case, the put, get, and remove operations have constant time complexity **O(1)**. However, in the worst case (due to collisions), the time complexity can degrade to **O(n)**, where n is the number of elements.
- **Space Complexity**: HashMap needs to store both keys and values, as well as additional overhead for the hash table and linked lists/tree structures used to resolve collisions.

Handling Collisions:

When two keys have the same hash code (a collision occurs), HashMap uses a linked list to store multiple values at the same index. In Java 8 and later, if the number of items in a bucket exceeds a certain threshold (default is 8), the linked list is replaced by a balanced binary tree (red-black tree) to improve performance, ensuring that collision resolution has **O(log n)** time complexity instead of **O(n)**.

Important Points about HashMap:

1. **Not Thread-Safe**: HashMap is not synchronized, meaning it is not safe to use in a concurrent/multi-threaded environment without external synchronization. For thread-safe maps, you can use ConcurrentHashMap.
2. **Load Factor and Rehashing**: HashMap dynamically increases the size of the hash table when the number of elements exceeds the load factor (default is 0.75). This process is called **rehashing** and involves redistributing all entries to a new, larger hash table.
3. **Allows Null Keys and Values**: HashMap allows one null key and multiple null values. However, it's generally advisable to avoid using null keys/values in maps.

Custom Objects as Keys:

To use custom objects as keys in a HashMap, you must ensure that the hashCode() and equals() methods are properly overridden, as HashMap relies on these methods to determine key equality and the appropriate bucket for storing the object.

Example of Custom Key in HashMap:

```
import java.util.HashMap;
import java.util.Objects;

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

```java

    // Constructor
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Override equals and hashCode methods
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class CustomHashMapKeyExample {
    public static void main(String[] args) {
        // Create a HashMap with custom Person objects as keys
        HashMap<Person, String> hashMap = new HashMap<>();

        // Add entries to the map
        hashMap.put(new Person("John", 25), "Engineer");
        hashMap.put(new Person("Alice", 30), "Doctor");

        // Retrieve a value based on the key
        Person person = new Person("John", 25);
        String profession = hashMap.get(person);
        System.out.println("John's profession: " + profession);
    }
}
```

Output:

```
John's profession: Engineer
```

When to Use HashMap:

- You need to associate keys with values, and fast retrieval of values based on keys is a priority.
- Duplicates are not allowed for keys, but you can have duplicate values.
- You don’t care about the order of elements.

Conclusion:

HashMap is an efficient data structure for storing key-value pairs, offering fast access, insertion, and deletion. It is widely used when there is no need for ordering of elements, and duplicates are not required. However, for multithreading, you may need ConcurrentHashMap, or you may synchronize a HashMap using Collections.synchronizedMap().

- **Internal Implementation**:
    - **Backing structure**: Hash table.
    - **Time complexity**: Constant time (O(1)) for get and put, assuming good hash distribution.
    - **Collisions**: Resolved by **separate chaining** (array of linked lists or trees).
    - **Load factor**: Rehashing occurs when the map is 75% full.
- **Use Cases**:
    - When you need **fast lookups** by key and order does not matter.

Map<String	Integer> map = new HashMap<>();
map.put("Apple"	1);
map.put("Banana"	2);

b) LinkedHashMap

LinkedHashMap in Java is a subclass of HashMap that maintains the **insertion order** or **access order** of the entries. This means that, unlike HashMap, which doesn't guarantee any specific order of elements, LinkedHashMap ensures that the elements are iterated in the order they were inserted or in the order they were accessed (if configured to do so).

Key Features of LinkedHashMap:

1. **Maintains Insertion Order**: By default, LinkedHashMap maintains the order in which key-value pairs are inserted.
2. **Optional Access Order**: If configured (by setting accessOrder to true), it can maintain access order, meaning it reorders entries based on when they were last accessed.
3. **Null Keys and Values**: Like HashMap, LinkedHashMap allows one null key and multiple null values.
4. **Non-Synchronized**: It is not synchronized, meaning it is not thread-safe. If you need synchronization, you can use Collections.synchronizedMap() or ConcurrentHashMap.

Internal Structure:

LinkedHashMap uses a **doubly linked list** that runs through all of its entries, in addition to the hash table used by HashMap. This linked list defines the iteration order.

Common Methods:

- put(K key, V value): Inserts the specified key-value pair into the map.
- get(Object key): Retrieves the value associated with the specified key.
- remove(Object key): Removes the key-value pair associated with the specified key.
- containsKey(Object key): Checks if the map contains the specified key.
- keySet(): Returns a Set view of the keys in the map.
- values(): Returns a Collection view of the values in the map.
- entrySet(): Returns a Set view of the mappings (key-value pairs).
- isEmpty(): Checks if the map contains no key-value mappings.

- size(): Returns the number of key-value mappings in the map.

Example of LinkedHashMap:

```
import java.util.LinkedHashMap;
import java.util.Map;


    public static void main(String[] args) {
        // Create a LinkedHashMap
        LinkedHashMap<String, Integer> linkedHashMap = new LinkedHashMap<>();

        // Add key-value pairs to the LinkedHashMap
        linkedHashMap.put("Apple", 10);
        linkedHashMap.put("Banana", 20);
        linkedHashMap.put("Orange", 30);
        linkedHashMap.put("Mango", 40);

        // Display the LinkedHashMap
        System.out.println("LinkedHashMap: " + linkedHashMap);

        // Access a value from the map\

        System.out.println("Value for key 'Banana': " + linkedHashMap.get("Banana"));

        // Remove a key-value pair
        linkedHashMap.remove("Mango");
        System.out.println("After removing 'Mango': " + linkedHashMap);

        // Iterate over the entries in the LinkedHashMap
        System.out.println("Entries in LinkedHashMap:");
        for (Map.Entry<String, Integer> entry: linkedHashMap.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Output:

```
LinkedHashMap: {Apple=10, Banana=20, Orange=30, Mango=40}
Value for key 'Banana': 20
After removing 'Mango': {Apple=10, Banana=20, Orange=30}
Entries in LinkedHashMap:
Apple -> 10
Banana -> 20
Orange -> 30
```

Constructors:

- **LinkedHashMap()**: Creates a new empty LinkedHashMap with default capacity (16) and load factor (0.75).
- **LinkedHashMap(int initialCapacity)**: Creates a LinkedHashMap with a specified initial capacity.
- **LinkedHashMap(int initialCapacity, float loadFactor)**: Creates a LinkedHashMap with a specified initial capacity and load factor.
- **LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)**: This constructor allows specifying the accessOrder. If accessOrder is true, the order is based on access; otherwise, it’s based on insertion.

Insertion Order vs Access Order:

By default, LinkedHashMap maintains insertion order. However, you can configure it to maintain **access order**, meaning the iteration order reflects the order in which entries were accessed (retrieved using get() or put()). To do this, use the following constructor:

```
LinkedHashMap<K, V> linkedHashMap = new LinkedHashMap<>(160.75f true);
```

Here, true indicates that the access order should be maintained instead of insertion order.

Example of Access Order:

```
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapAccessOrderExample {
    public static void main(String[] args) {
        // Create a LinkedHashMap with accessOrder set to true
        LinkedHashMap<String, Integer> linkedHashMap = new LinkedHashMap<>(16, 0.75f, true);

        // Add key-value pairs to the LinkedHashMap
        linkedHashMap.put("Apple", 10);
        linkedHashMap.put("Banana", 20);
        linkedHashMap.put("Orange", 30);

        // Access the 'Apple' entry
        linkedHashMap.get("Apple");

        // Iterate over the LinkedHashMap
        System.out.println("Entries in LinkedHashMap:");
        for (Map.Entry<String, Integer> entry : linkedHashMap.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Output (Access Order):

```
Entries in LinkedHashMap:
Banana -> 20
Orange -> 30
Apple -> 10
```

In this example, the entry for "Apple" appears last because it was accessed after the other entries.

Performance of LinkedHashMap:

- **Time Complexity**: Similar to HashMap, basic operations like put(), get(), and remove() have constant time complexity **O(1)**.
- **Space Complexity**: LinkedHashMap uses more memory than HashMap because it maintains a linked list to preserve insertion or access order.

When to Use LinkedHashMap:

- You need to maintain the **order of insertion** while having the performance benefits of a HashMap.
- You want to maintain the **order of access** to entries for an LRU (Least Recently Used) cache or similar functionality.

Example Use Case: LRU Cache

An LRU (Least Recently Used) cache can be implemented using LinkedHashMap by setting accessOrder to true and overriding the removeEldestEntry() method to specify when the eldest entry should be removed.

```java
import java.util.LinkedHashMap;
import java.util.Map;

class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75f, true);
        this.capacity = capacity;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capacity;
    }
}

public class LRUCacheExample {
    public static void main(String[] args) {
        LRUCache<Integer, String> cache = new LRUCache<>(3);

        cache.put(1, "A");
        cache.put(2, "B");
        cache.put(3, "C");
        System.out.println("Cache: " + cache);

        // Access entry 2
        cache.get(2);
        System.out.println("After accessing key 2: " + cache);

        // Add a new entry, which causes the eldest entry to be removed
        cache.put(4, "D");
        System.out.println("After adding key 4: " + cache);
    }
}
```

Output:

```
Cache: {1=A, 2=B, 3=C}
After accessing key 2: {1=A, 3=C, 2=B}
After adding key 4: {3=C, 2=B, 4=D}
```

In this example, the LRU cache evicts the oldest entry when the cache reaches its capacity.

Conclusion:

LinkedHashMap is a useful data structure when you need to maintain the insertion or access order of elements while taking advantage of the fast performance of a hash table. It's particularly beneficial for use cases such as building LRU caches or any scenario where order-sensitive iteration is required.

- **Internal Implementation**:
  - Same as HashMap, but **maintains insertion order** by using a doubly linked list.
  - Slightly slower than HashMap due to order maintenance.

- **Use Cases**:
  - When you need fast lookups and want to **preserve insertion order**.

Map<String	Integer> map = new LinkedHashMap<>();
map.put("Apple"	1);
map.put("Banana"	2);

c) TreeMap

In Java, a TreeMap is a part of the Java Collections Framework that implements the NavigableMap interface and is based on a Red-Black tree structure. Unlike HashMap and LinkedHashMap, TreeMap stores key-value pairs in **sorted order** based on the natural ordering of the keys or by a custom comparator provided at map creation time.

Key Features of TreeMap:

1. **Sorted Map**: It sorts the keys in ascending order by default (according to their natural ordering or based on a comparator).
2. **NavigableMap Implementation**: TreeMap implements the NavigableMap interface, providing methods to navigate through the map (like lowerEntry(), higherEntry(), subMap(), etc.).
3. **No null keys**: TreeMap does **not allow null keys**. Attempting to insert a null key will throw a NullPointerException. However, it does allow multiple null values.
4. **Performance**: TreeMap guarantees **logarithmic time complexity (O(log n))** for basic operations such as put(), get(), remove(), etc., due to the underlying Red-Black tree structure.
5. **Non-synchronized**: TreeMap is not synchronized, so it's not thread-safe. If thread safety is required, Collections.synchronizedMap() can be used.

Constructors:

- **TreeMap()**: Constructs an empty TreeMap that sorts its entries according to the natural ordering of the keys.
- **TreeMap(Comparator<? super K> comparator)**: Constructs a new TreeMap that sorts according to the specified comparator.
- **TreeMap(Map<? extends K, ? extends V> m)**: Constructs a new TreeMap with the same mappings as the specified Map, sorted according to the natural ordering of its keys.
- **TreeMap(SortedMap<K, ? extends V> m)**: Constructs a new TreeMap with the same mappings and the same order as the specified sorted map.

Example of TreeMap:
import java.util.TreeMap;
import java.util.Map;
public class TreeMapExample {
public static void main(String[] args) {
// Create a TreeMap
TreeMap<String, Integer> treeMap = new TreeMap<>();
// Add key-value pairs to the TreeMap
treeMap.put("Apple", 50);
treeMap.put("Banana", 20);
treeMap.put("Orange", 30);
treeMap.put("Mango", 10);

```java

        // Display the TreeMap (Sorted in natural order)
        System.out.println("TreeMap: " + treeMap);

        // Get the value associated with a specific key
        System.out.println("Value for key 'Banana': " + treeMap.get("Banana"));

        // Remove a key-value pair
        treeMap.remove("Mango");
        System.out.println("After removing 'Mango': " + treeMap);

        // Iterate over the entries in the TreeMap
        System.out.println("Entries in TreeMap:");
        for (Map.Entry<String, Integer> entry : treeMap.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }

        // Get the first and last entry in the TreeMap
        System.out.println("First Entry: " + treeMap.firstEntry());
        System.out.println("Last Entry: " + treeMap.lastEntry());
    }
}
```

Output:

```
TreeMap: {Apple=50, Banana=20, Mango=10, Orange=30}
Value for key 'Banana': 20
After removing 'Mango': {Apple=50, Banana=20, Orange=30}
Entries in TreeMap:
Apple -> 50
Banana -> 20
Orange -> 30
First Entry: Apple=50
Last Entry: Orange=30
```

Common Methods of TreeMap:

1. **put(K key, V value)**: Inserts the specified key-value pair.
2. **get(Object key)**: Returns the value associated with the specified key.
3. **remove(Object key)**: Removes the mapping for the specified key.
4. **firstKey()**: Returns the first (lowest) key in the map.
5. **lastKey()**: Returns the last (highest) key in the map.
6. **firstEntry()**: Returns a key-value mapping associated with the lowest key.
7. **lastEntry()**: Returns a key-value mapping associated with the highest key.
8. **subMap(K fromKey, K toKey)**: Returns a view of the portion of this map whose keys range from fromKey to toKey.
9. **higherKey(K key)**: Returns the least key strictly greater than the given key.
10. **lowerKey(K key)**: Returns the greatest key strictly less than the given key.
11. **descendingMap()**: Returns a reverse order view of the map.

Custom Sorting in TreeMap:

By default, TreeMap sorts keys in their natural order (ascending order for numbers, alphabetical order for strings). If you want to sort the keys in a custom order, you can provide a custom comparator when creating the TreeMap.

Example of Custom Sorting (Descending Order):

```
import java.util.Comparator;
import java.util.TreeMap;

public class CustomTreeMapSorting {
    public static void main(String[] args) {
        // Custom comparator to sort keys in descending order
        TreeMap<String, Integer> treeMap = new TreeMap<>(Comparator.reverseOrder());

        // Add key-value pairs to the TreeMap
        treeMap.put("Apple", 50);
        treeMap.put("Banana", 20);
        treeMap.put("Orange", 30);
        treeMap.put("Mango", 10);

        // Display the TreeMap (Sorted in reverse order)
        System.out.println("TreeMap in Descending Order: " + treeMap);
    }
}
```

Output:

```
TreeMap in Descending Order: {Orange=30, Mango=10, Banana=20, Apple=50}
```

Performance:

- **Time Complexity**: Since TreeMap is implemented as a Red-Black tree, the time complexity for the basic operations (e.g., put(), get(), remove()) is **O(log n)**.
- **Space Complexity**: The space complexity is proportional to the number of key-value mappings.

When to Use TreeMap:

- When you need to maintain a **sorted order** of the keys.
- When you require **range view** methods like subMap(), headMap(), or tailMap().
- When you need methods to **navigate** the keys (e.g., higherKey(), lowerKey()).

Example Use Case: Navigating a TreeMap

You can use TreeMap when you need to access entries based on their relative position in the sorted order:

import java.util.TreeMap;

public

```
class TreeMapNavigationExample {
    public static void main(String[] args) {
        TreeMap<Integer, String> treeMap = new TreeMap<>();

        // Add key-value pairs to the TreeMap
        treeMap.put(1, "One");
        treeMap.put(3, "Three");
        treeMap.put(2, "Two");
        treeMap.put(5, "Five");

        // Find the least key greater than 2
        System.out.println("Key greater than 2: " + treeMap.higherKey(2));

        // Find the greatest key less than 3
        System.out.println("Key less than 3: " + treeMap.lowerKey(3));

        // Get a subMap between 2 and 5
        System.out.println("SubMap (2 to 5): " + treeMap.subMap(2, 5));
    }
}
```

Output:

```
vbnet
Copy code
Key greater than 2: 3
Key less than 3: 2
SubMap (2 to 5): {2=Two, 3=Three}
```

Conclusion:

TreeMap is an ideal choice when you need a map that stores elements in **sorted order** and provides navigation methods. It offers guaranteed log-time performance for most operations, making it efficient for range queries and key ordering operations.

- **Internal Implementation**:
  - **Backing structure**: Red-Black Tree.
  - **Time complexity**: Logarithmic time (O(log n)) for get, put, remove.
  - **Order**: Sorted by keys, either natural ordering or custom comparator.
- **Use Cases**:
  - When you need a map with **sorted keys**.

Map<String	Integer> map = new TreeMap<>();
map.put("Apple"	1);
map.put("Banana"	2);

d) ConcurrentHashMap

- **Internal Implementation**:
  - Similar to HashMap but provides **thread-safe** operations.
  - Utilizes **lock stripping** to improve performance in multi-threaded environments.
- **Use Cases**:
  - When you need a thread-safe map in a **concurrent environment**.

```
Map<String, Integer> map = new ConcurrentHashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);

```

4. Queue Interface

A Queue is a collection designed for holding elements prior to processing. It typically follows **FIFO (First-In-First-Out)** order.

Implementations of Queue:

- **LinkedList** (implements Deque)
- **PriorityQueue**
- **ArrayDeque**

a) PriorityQueue

- **Internal Implementation**:
  - Backed by a binary heap, which allows **natural ordering** or custom comparators.
  - **Time complexity**: Logarithmic time (O(log n)) for insertion and removal based on priority.
- **Use Cases**:
  - When you need to process elements based on **priority** rather than the order they were added.

```
Queue<Integer> queue = new PriorityQueue<>();
queue.add(10);
queue.add(20);
```

b) ArrayDeque

- **Internal Implementation**:
  - Resizable array-based implementation of Deque (double-ended queue).
  - Provides fast insertion/removal at both ends.
- **Use Cases**:
  - When you need a **stack** (LIFO) or **queue** (FIFO) behavior with better performance than LinkedList.

```
Deque<String> deque = new ArrayDeque<>();
deque.addFirst("Apple");
deque.addLast("Banana");
```

Summary of Use Cases:

- **ArrayList**: When random access and frequent read operations are required.
- **LinkedList**: When frequent insertions and deletions are needed.
- **HashSet**: For ensuring unique elements, with fast lookups and no specific order.
- **TreeSet**: When you need a sorted set of unique elements.
- **HashMap**: When fast access to key-value pairs is required, with no specific ordering.

- **TreeMap**: When you need a sorted map by keys.
- **PriorityQueue**: When processing elements based on priority is important.
- **ConcurrentHashMap**: For thread-safe map access in concurrent applications.

Note: **Everything that start with capital letter is a Class ArrayList, String etc.**

Note: /0 - it is null character, its ASCII value is 0 (Zero), it is like a terminator it tells where string is ended.

✹ Iterate over Map

Yes, you can iterate over a HashMap in Java. Since a HashMap stores key-value pairs, there are several ways to iterate over it depending on whether you want to iterate over the keys, the values, or both. Below are the common approaches:

1. Iterating over the Entry (Key-Value Pairs)

You can use the entrySet() method to get a Set of key-value pairs, and then iterate over the set.

Example:

```
import java.util.HashMap;
import java.util.Map;

public class HashMapIteration {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Apple", 3);
        map.put("Banana", 2);
        map.put("Orange", 4);

        // Iterate over key-value pairs (Map.Entry)
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
        }
    }
}
```

2. Iterating over the Keys

If you're only interested in the keys, you can use the keySet() method to get a set of all the keys.

Example:

```
public class HashMapKeyIteration {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Apple", 3);
        map.put("Banana", 2);
        map.put("Orange", 4);

        // Iterate over keys
        for (String key : map.keySet()) {
            System.out.println("Key: " + key);
        }
    }
}
```

3. Iterating over the Values

If you only need the values, you can use the values() method to get a collection of all the values in the HashMap.

Example:

```
public class HashMapValueIteration {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Apple", 3);
        map.put("Banana", 2);
        map.put("Orange", 4);

        // Iterate over values
        for (Integer value : map.values()) {
            System.out.println("Value: " + value);
        }
    }
}
```

4. Using Java 8's forEach() with Lambda Expression

Java 8 introduced the forEach() method, which makes it easy to iterate over a HashMap using a lambda expression.

Example:

```
public class HashMapForEachIteration {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Apple", 3);
        map.put("Banana", 2);
        map.put("Orange", 4);

        // Iterate using forEach and lambda expression
        map.forEach((key, value) -> System.out.println("Key: " + key + ", Value: " + value));
    }
}
```

Summary of Iteration Methods:

- **entrySet()**: Iterates over both keys and values (Map.Entry).
- **keySet()**: Iterates only over the keys.
- **values()**: Iterates only over the values.
- **forEach()**: Provides a modern way to iterate using a lambda expression.

Each method is useful in different scenarios depending on what part of the HashMap you need to access.

Iterating Over Both LinkedHashMap and TreeMap:

You can use all the common iteration techniques (entrySet(), keySet(), values(), forEach(), etc.) on LinkedHashMap and TreeMap just like with HashMap. Here's an example showing both maps:

Example:

```
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.TreeMap;

public class MapIteration {
    public static void main(String[] args) {
        // LinkedHashMap maintains insertion order
        LinkedHashMap<String, Integer> linkedMap = new LinkedHashMap<>();
        linkedMap.put("Apple", 3);
        linkedMap.put("Banana", 2);
        linkedMap.put("Orange", 5);

        // TreeMap maintains natural ordering (alphabetical for String keys)
        TreeMap<String, Integer> treeMap = new TreeMap<>();
        treeMap.put("Banana", 2);
        treeMap.put("Apple", 3);
        treeMap.put("Orange", 5);

        System.out.println("LinkedHashMap (Insertion Order):");
        linkedMap.forEach((key, value) -> System.out.println("Key: " + key + ", Value: " + value));

        System.out.println("\nTreeMap (Natural Order):");
        treeMap.forEach((key, value) -> System.out.println("Key: " + key + ", Value: " + value));
    }
}
```

Output:

```
LinkedHashMap (Insertion Order):
Key: Apple, Value: 3
Key: Banana, Value: 2
Key: Orange, Value: 5

TreeMap (Natural Order):
Key: Apple, Value: 3
Key: Banana, Value: 2
Key: Orange, Value: 5
```

Summary:

- LinkedHashMap maintains insertion order, and you can iterate over it the same way as a HashMap.
- TreeMap maintains natural ordering (ascending order by default), and you can iterate over it similarly.
- The iteration techniques (entrySet(), keySet(), values(), forEach()) work for all types of Map, including HashMap, LinkedHashMap, and TreeMap.