

1. Eight basic data type

Here's a table summarizing the eight basic data types in Java:

Data Type	Size	Range	Use
byte	8 bits	-128 to 127	Saves memory in large arrays or binary data.
short	16 bits	-32,768 to 32,767	Useful for saving memory in large arrays.
int	32 bits	-2,147,483,648 to 2,147,483,647	Default type for integers.
long	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	For larger integer values.
float	32 bits	$\pm 3.40282347\text{E}+38\text{F}$ (7 decimal digits)	For single-precision floating-point numbers.
double	64 bits	$\pm 1.79769313486231570\text{E}+308$ (15 decimal digits)	Default type for decimals, double precision.
char	16 bits	'\u0000' (0) to '\uffff' (65,535)	Stores a single character or Unicode value.
boolean	1 bit (conceptual)	true or false	For logical (true/false) values.

Autoboxing and **Boxing** in Java relate to the conversion between **primitive data types** and their corresponding **wrapper classes**. Here's an explanation:

1. Boxing

- **Definition:** The manual conversion of a primitive data type into its corresponding wrapper class object.
- **Example:**

```
int num = 5; // primitive type
Integer boxedNum = Integer.valueOf(num); // Boxing
```
- **Explanation:** Here, `Integer.valueOf(num)` is explicitly used to convert the primitive `int` to the `Integer` object.

2. Autoboxing

- **Definition:** The automatic conversion of a primitive data type into its corresponding wrapper class object by the compiler.
- **Example:**

```
int num = 10; // primitive type
Integer autoBoxedNum = num; // Autoboxing
```
- **Explanation:** Here, the compiler automatically converts `int num` to an `Integer` object without requiring `Integer.valueOf()`.

Key Differences

Aspect	Boxing	Autoboxing
Conversion	Manual	Automatic
When Used	Explicitly invoked by the programmer.	Handled by the Java compiler.
Example	<code>Integer.valueOf(primitiveValue)</code>	<code>Integer obj = primitiveValue;</code>

Unboxing

The reverse of boxing, where an object of a wrapper class is converted back to its corresponding primitive type.

Example of Autoboxing and Unboxing:

```
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        // Autoboxing
        ArrayList<Integer> list = new ArrayList<>();
        list.add(25); // Compiler automatically converts int to Integer

        // Unboxing
        int value = list.get(0); // Compiler automatically converts Integer to int
        System.out.println(value);
    }
}
```

- String/StringBuilder/StringBuffer

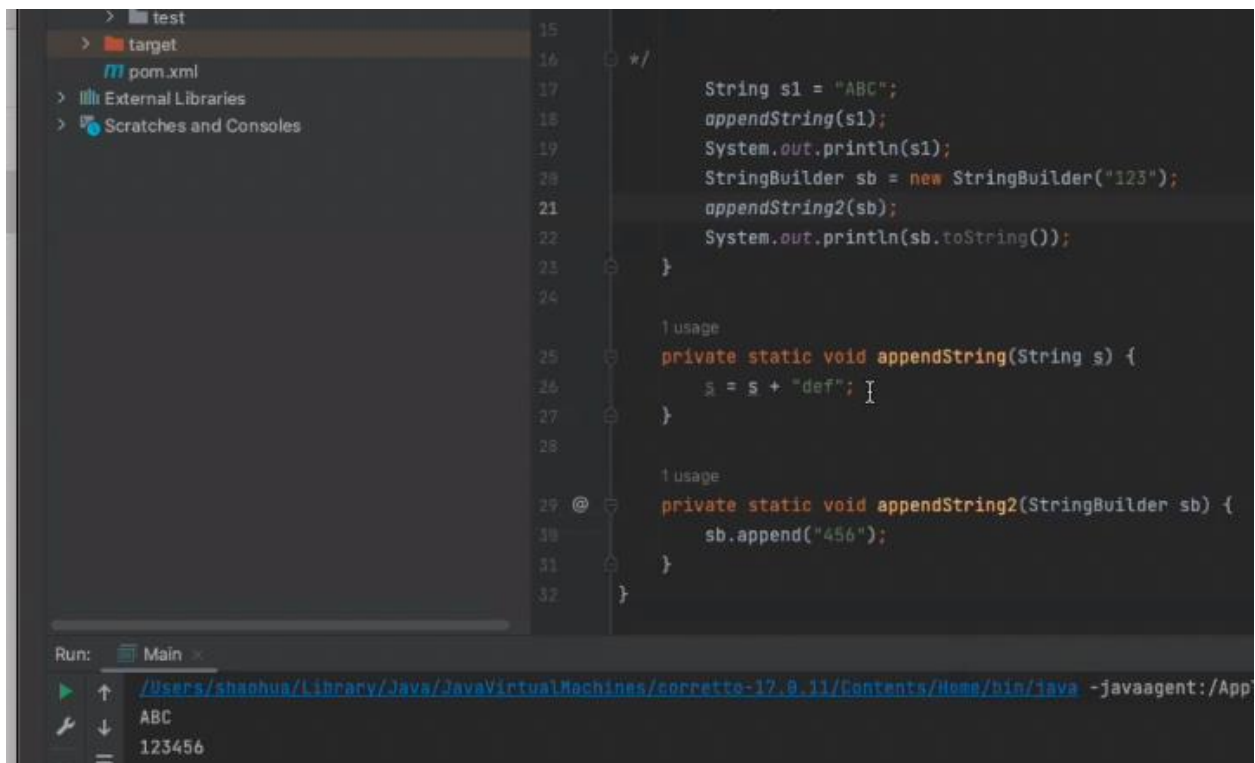
Here's a comparison of **String**, **StringBuilder**, and **StringBuffer** in Java:

Aspect	String	StringBuilder	StringBuffer
Mutability	Immutable	Mutable	Mutable
Thread-Safety	Not thread-safe	Not thread-safe	Thread-safe
Performance	Slower in operations involving string manipulation due to immutability.	Faster for single-threaded string operations.	Slower than StringBuilder due to synchronization.
Use Case	For fixed, constant, or read-only strings.	For string manipulations in single-threaded environments.	For string manipulations in multi-threaded environments.

Synchronization	No	No	Yes
Memory Usage	Creates a new object for each modification.	Operates on the same object, reducing memory usage.	Operates on the same object, reducing memory usage.
Methods for Modification	Not modifiable. Creates a new string for each operation like concat(), replace().	Modifiable using methods like append(), insert(), delete(), reverse().	Same as StringBuilder.
Example Usage	<pre>```java String s = "Hello"; s = s.concat(" World");</pre>	<pre>```java StringBuilder sb = new StringBuilder("Hello"); sb.append(" World");</pre>	<pre>```java StringBuffer sb = new StringBuffer("Hello"); sb.append(" World");</pre>
Output	s: "Hello World"	sb: "Hello World"	sb: "Hello World"

Key Takeaways

- Use **String** for immutable or read-only string data.
- Use **StringBuilder** for better performance when working with mutable strings in single-threaded environments.
- Use **StringBuffer** for thread-safe string operations, although it's less common in modern Java programming as synchronization can be manually handled if needed.

A screenshot of an IDE window. The top pane shows a Java file with the following code:

```
15  
16  
17     String s1 = "ABC";  
18     appendString(s1);  
19     System.out.println(s1);  
20  
21     StringBuilder sb = new StringBuilder("123");  
22     appendString2(sb);  
23     System.out.println(sb.toString());  
24  
25 }  
26  
27 } usage  
28  
29 private static void appendString(String s) {  
30     s = s + "def";  
31 }  
32  
33 } usage  
34  
35 @ private static void appendString2(StringBuilder sb) {  
36     sb.append("456");  
37 }  
38  
39 }
```

The bottom pane shows the output of the program:

```
Run: Main  
ABC  
123456
```

String/Integer intern pool

Intern Pool in Java

The intern pool is a special pool of memory used to store immutable objects like **Strings** and, to some extent, **Integer** objects in Java. Here's how the intern pool works for **String** and **Integer**:

1. String Intern Pool

- **What it is:** A part of the heap memory that stores unique instances of String literals. This pool ensures that identical String objects share the same memory location to save memory and improve performance.
- **How it works:**
 - When a String literal is created (e.g., String s = "hello";), Java checks if the literal already exists in the pool.
 - If it exists, the reference to the existing literal is returned.
 - If it does not exist, a new literal is added to the pool.
 - Explicitly calling String.intern() can also add a String to the pool.

- **Example:**

String s1 = "hello"; // Created in the String pool.

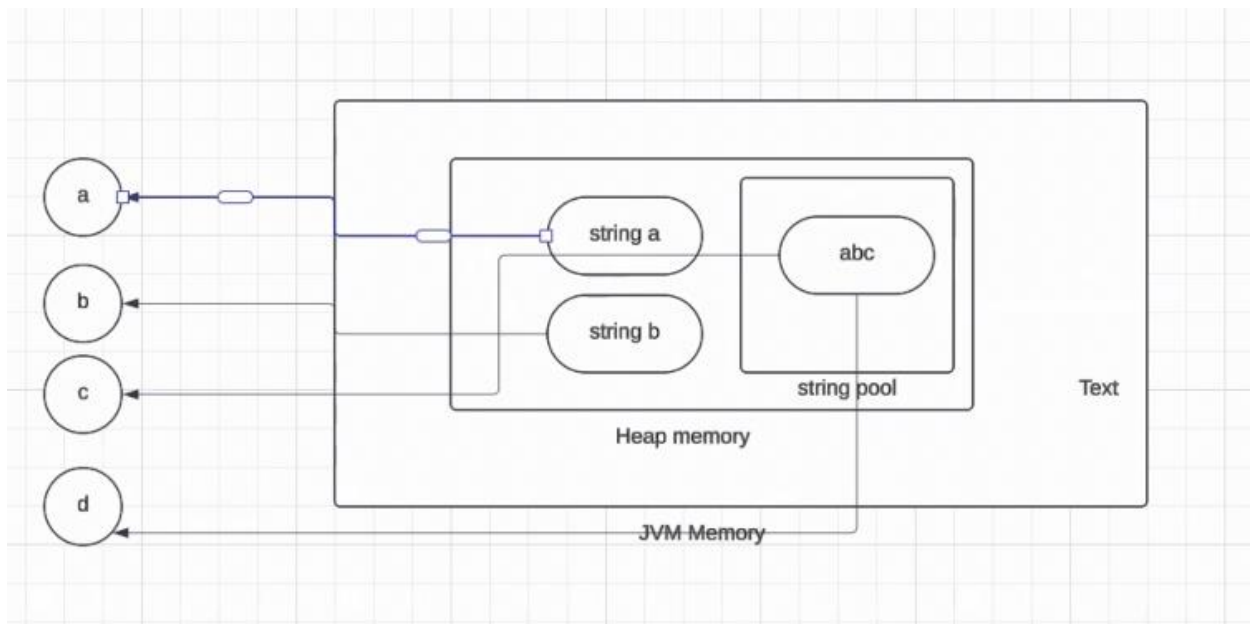
String s2 = "hello"; // Points to the same "hello" in the pool.

String s3 = new String("hello"); // Created in the heap, not the pool.

System.out.println(s1 == s2); // true (same reference)

System.out.println(s1 == s3); // false (different references)

System.out.println(s1 == s3.intern()); // true (interned string matches the pool)



```
String a = new String( original: "abc");  
String b = new String( original: "abc");  
String c = "abc";  
String d = "abc";
```

```
System.out.println(a == b); // false  
System.out.println(a == c); // false  
System.out.println(c == d); // true
```

2. Integer Intern Pool

- **What it is:** The Integer class uses an internal **cache** for commonly used integers to optimize memory and performance.
- **How it works:**
 - For autoboxing of integers within the range **-128 to 127**, Java uses cached Integer objects.
 - For integers outside this range, a new Integer object is created.
- **Example:**

```
Integer i1 = 100; // Cached
Integer i2 = 100; // Points to the same cached object.
Integer i3 = 200; // New object outside the cache range.
Integer i4 = 200; // Another new object outside the cache range.

System.out.println(i1 == i2); // true (same cached reference)
System.out.println(i3 == i4); // false (different references)
```
- **Customizing Cache Range:** The integer cache range can be customized by setting the JVM argument `-XX:AutoBoxCacheMax=<value>`. This affects how large the cache range is.

```

Integer aa = 15;
Integer bb = 15;
Integer dd = Integer.valueOf(15);
System.out.println(aa == bb); // true
System.out.println(aa == dd); // true

Integer ee = 300;
Integer ff = 300;
System.out.println(ee == ff); // false

```

Key Differences

Aspect	String Intern Pool	Integer Cache
Scope	All String literals are added to the pool.	Limited to integers in the range -128 to 127 (by default).
Customizability	Not configurable by the user.	Cache range can be customized via JVM options.
Thread-Safety	Thread-safe.	Thread-safe.
Usage	Optimizes memory for repeated strings.	Optimizes memory for small integer values.

Key Takeaways

- Use `String.intern()` to explicitly add or retrieve strings from the intern pool.
- Rely on autoboxing for efficient Integer usage within the default cache range.

"==" vs `equals()`

Here's a comparison between `==` and `equals()` in Java:

Feature	<code>==</code> Operator	<code>equals()</code> Method
---------	--------------------------	------------------------------

Purpose	Compares references (memory addresses) for objects or values for primitives.	Compares the content (or state) of objects.
Primitive Types	Directly compares values. For example: <code>int a = 5; int b = 5; (a == b)</code> is true.	Cannot be used with primitive types.
Object References	Checks if two references point to the same object in memory.	Checks if two objects are logically equivalent based on their content.
Default Behavior	For objects, compares memory locations. Example: <code>String s1 = new String("abc"); String s2 = new String("abc"); (s1 == s2)</code> is false.	The <code>Object</code> class's <code>equals()</code> method behaves like <code>==</code> unless overridden. Many classes (like <code>String</code>) override it for logical equality.
Custom Logic	Cannot be overridden.	Can be overridden in custom classes to define logical equivalence.
String Example	For strings, <code>==</code> checks reference equality. Example: <code>String a = "abc"; String b = "abc"; (a == b)</code> is true due to string interning, but <code>new String("abc") == new String("abc")</code> is false.	Compares content of strings. Example: <code>new String("abc").equals(new String("abc"))</code> is true.
Null Handling	Can compare null safely (e.g., <code>obj == null</code> is valid).	Throws <code>NullPointerException</code> if <code>equals()</code> is called on null.

Summary

- Use `==` to check reference equality (whether two references point to the same object).
- Use `equals()` to check logical equality (whether two objects are meaningfully equivalent).

Equals() vs hashCode()

Here's a comparison between `equals()` and `hashCode()` in Java:

Feature	<code>equals()</code> Method	<code>hashCode()</code> Method
Purpose	Determines whether two objects are logically equal based on their content or state.	Returns an integer hash code that represents the object for hash-based data structures like HashMap.
Default Behavior	Defined in the Object class to compare memory addresses (like ==) . Usually overridden for custom logic.	Defined in the Object class to return a unique identifier for the object based on memory address.
Customization	Can be overridden to define custom equality logic.	Must be overridden whenever equals() is overridden to maintain the contract between them.
Contract with Each Other	If two objects are equal (as per <code>equals()</code>), they must have the same hash code.	If two objects have the same hash code, they may or may not be equal.
Use in Hash-Based Collections	Used to check logical equality of keys in hash-based collections like HashMap, HashSet.	Used to determine the bucket location of objects in hash-based collections like HashMap, HashSet.
Performance Role	Slower, as it compares the content or state of objects.	Faster, as it computes a single integer value for quick lookups in hash-based structures.
Example Use Case	To compare if two objects are logically the same (e.g., <code>person1.equals(person2)</code>).	Used in hash-based structures for efficient storage and retrieval.

Important Rules

1. Consistency:

- If `a.equals(b)`, then `a.hashCode() == b.hashCode()` must always be true.
- If `a.hashCode() != b.hashCode()`, then `a.equals(b)` must always be false.

2. Overrides:

- Always override hashCode() if you override equals().
- Failure to do so may result in unexpected behavior in hash-based collections.

Example Code

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

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

    @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 && name.equals(person.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age); // Combines fields into a hash code
    }
}
```

Summary

- equals(): Defines logical equality between objects.
- hashCode(): Provides a hash code for the object, critical for hash-based collections.
- Always maintain the equals() and hashCode() contract to ensure proper behavior in collections.

Data structure

Collection vs Collections

In Java, **Collection** and **Collections** are often confused, but they serve different purposes. Here's a comparison:

Feature	Collection	Collections
Definition	Collection is an interface in the java.util package, part of the Java Collections Framework.	Collections is a utility class in the java.util package, providing static methods to operate on collections.
Type	Interface	Class
Purpose	Defines a general contract for working with collections like List, Set, Queue.	Provides utility methods for common operations like sorting, searching, reversing, etc.
Examples	Implemented by classes such as ArrayList, HashSet, LinkedList.	Contains static methods like Collections.sort(), Collections.max(), Collections.unmodifiableList().
Inheritance	Part of the hierarchy: Collection -> Set, List, Queue.	Not part of the hierarchy; used as a helper class.
Usage Context	Used to define and manipulate the structure and behavior of a collection.	Used to perform common tasks on or modify existing collections.
Static Methods	Not applicable, as it's an interface.	Includes static methods like: - sort(List<T> list) (Sorts a list) - reverse(List<?> list) (Reverses a list) - synchronizedList(List<T> list) (Creates a thread-safe list)

Code Example

Using Collection

```
import java.util.ArrayList;
import java.util.Collection;

public class CollectionExample {
    public static void main(String[] args) {
        Collection<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

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

Using Collections

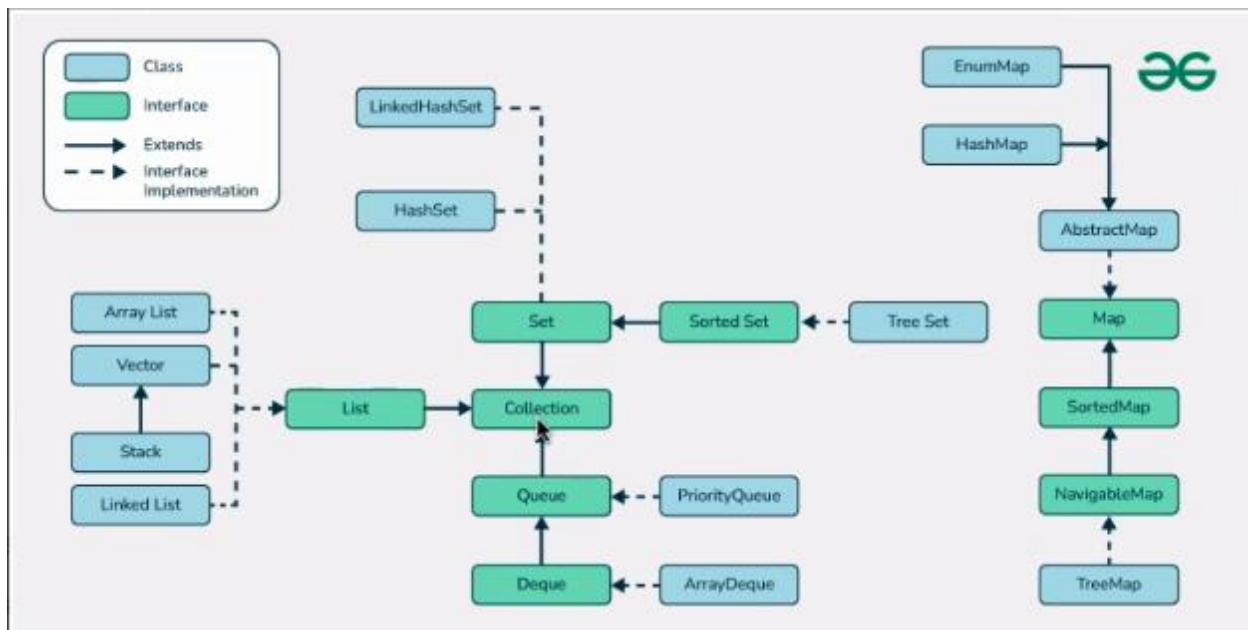
```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CollectionsExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(5);
        numbers.add(1);
        numbers.add(3);

        // Sort the list
        Collections.sort(numbers);

        // Reverse the list
        Collections.reverse(numbers);

        // Print the list
        System.out.println(numbers);
    }
}
```



Key Points

1. Collection:

- Defines basic collection operations (add(), remove(), size(), etc.).
- Is the root interface for all collection classes.

2. Collections:

- Contains static utility methods to work with collections.
- Examples: Sorting (sort()), Searching (binarySearch()), Synchronizing (synchronizedList()).

Tip: Think of Collection as the blueprint/interface for collection objects, and Collections as the toolbox for performing operations on those objects.

HashSet vs LinkedHashSet vs TreeSet

Here is a comparison of **HashSet**, **LinkedHashSet**, and **TreeSet** in Java:

Feature	HashSet	LinkedHashSet	TreeSet
Definition	Implements the Set interface,	Implements the Set interface, backed by a LinkedHashMap.	Implements the Set interface, backed by a TreeMap.

	backed by a HashMap.		
Order	No guarantee of iteration order.	Maintains insertion order.	Maintains elements in natural order or a custom comparator-defined order.
Performance (Add/Remove)	O(1) average-case performance.	O(1) average-case performance.	O(log n) , as it uses a balanced tree structure.
Null Handling	Allows one null element.	Allows one null element.	Does not allow null values (throws <code>NullPointerException</code>).
Duplicates	Does not allow duplicate elements.	Does not allow duplicate elements.	Does not allow duplicate elements.
Best Use Case	Fast lookups and inserts when order does not matter.	Maintaining order of elements without duplicates.	Maintaining sorted elements in natural or custom order.
Underlying Data Structure	Hash table.	Hash table with a doubly-linked list for ordering.	Red-Black tree.

Code Examples

HashSet Example

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> hashSet = new HashSet<>();
        hashSet.add("Apple");
        hashSet.add("Banana");
        hashSet.add("Cherry");

        System.out.println("HashSet: " + hashSet);
        // Output order may vary: [Banana, Cherry, Apple]
```

```
}  
}
```

LinkedHashSet Example

```
import java.util.LinkedHashSet;  
  
public class LinkedHashSetExample {  
    public static void main(String[] args) {  
        LinkedHashSet<String> linkedHashSet = new LinkedHashSet<>();  
        linkedHashSet.add("Apple");  
        linkedHashSet.add("Banana");  
        linkedHashSet.add("Cherry");  
  
        System.out.println("LinkedHashSet: " + linkedHashSet);  
        // Output: [Apple, Banana, Cherry]  
    }  
}
```

TreeSet Example

```
import java.util.TreeSet;  
  
public class TreeSetExample {  
    public static void main(String[] args) {  
        TreeSet<String> treeSet = new TreeSet<>();  
        treeSet.add("Apple");  
        treeSet.add("Banana");  
        treeSet.add("Cherry");  
  
        System.out.println("TreeSet: " + treeSet);  
        // Output: [Apple, Banana, Cherry] (sorted order)  
    }  
}
```

Key Differences

1. Order:

- **HashSet: Unordered.**
- **LinkedHashSet: Maintains insertion order.**

- **TreeSet: Sorted order.**

2. Performance:

- **HashSet and LinkedHashSet are faster than TreeSet for add/remove operations.**
- **TreeSet is preferred for sorted collections.**

3. Null Values:

- **HashSet and LinkedHashSet allow a single null value.**
- **TreeSet does not allow null.**

When to Use

- Use **HashSet** for quick operations and when order is not important.
- Use **LinkedHashSet** when you need to maintain the insertion order.
- Use **TreeSet** when you need to keep elements in sorted order.

LinkedList vs ArrayList

Here's a detailed comparison of **LinkedList** and **ArrayList** in Java:

Feature	ArrayList	LinkedList
Underlying Data Structure	Resizable array.	Doubly linked list.
Access Time	O(1) for accessing elements by index.	O(n) for accessing elements by index, as traversal is needed.
Insertion (Middle)	O(n) due to array resizing and shifting.	O(1) when using iterators, as only pointer adjustments are required.
Insertion (End)	Amortized O(1) (may require array resizing).	O(1) (no resizing required).

Deletion	O(n) for middle or start elements, as shifting occurs.	O(1) for middle or start elements, as only pointers are adjusted.
Memory Efficiency	More memory efficient, as it stores only the data.	Requires extra memory for node pointers.
Iteration Performance	Faster due to contiguous memory and better cache locality.	Slower due to pointer chasing.
Best Use Case	Use when random access is needed or frequent additions/removals are at the end.	Use when frequent additions/removals are in the middle or start.
Thread Safety	Not synchronized.	Not synchronized.
Implements	List, RandomAccess.	List, Deque, and Queue (can be used as a doubly-ended queue).
Insertion Order	Maintains insertion order.	Maintains insertion order.
Performance for Large Data	Performs better for fixed-size or rarely resized lists.	Performs better for frequently modified lists.

Code Examples

ArrayList Example

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> arrayList = new ArrayList<>();
        arrayList.add("Apple");
        arrayList.add("Banana");
        arrayList.add("Cherry");

        System.out.println("ArrayList: " + arrayList);
        // Access element
        System.out.println("Element at index 1: " + arrayList.get(1)); // O(1)
    }
}
```

```
}  
}
```

LinkedList Example

```
import java.util.LinkedList;  
  
public class LinkedListExample {  
    public static void main(String[] args) {  
        LinkedList<String> linkedList = new LinkedList<>();  
        linkedList.add("Apple");  
        linkedList.add("Banana");  
        linkedList.add("Cherry");  
  
        System.out.println("LinkedList: " + linkedList);  
        // Access element  
        System.out.println("Element at index 1: " + linkedList.get(1)); // O(n)  
    }  
}
```

Key Differences

1. Performance:

- **Access:** ArrayList is faster due to direct index-based access.
- **Insertion/Deletion:** LinkedList is faster for frequent operations in the middle or start.

2. Memory Usage:

- ArrayList uses less memory, as it only stores data.
- LinkedList uses more memory due to additional pointers for each node.

3. Use Case:

- Use ArrayList for scenarios requiring random access or appending elements.
- Use LinkedList for scenarios with frequent insertions/deletions at various positions.

When to Use

- **ArrayList**: When random access is frequent, and you primarily append elements.
- **LinkedList**: When insertions and deletions occur frequently in the middle or beginning of the list.

Queue: PriorityQueue->heap

PriorityQueue in Java and Its Underlying Heap Structure

Overview of PriorityQueue

- **PriorityQueue** is part of the **Java Collections Framework**, used to implement a priority heap.
- It maintains its elements in a specific order based on their **natural ordering** (if they are Comparable) or a **custom comparator** provided at construction.
- The queue operates on the principle that the **head** of the queue is the **smallest element** (min-heap by default).

Key Characteristics

Feature	Description
Data Structure	Uses a binary heap as its underlying data structure.
Order of Elements	Elements are ordered based on natural ordering or a custom comparator .
Access Type	Does not support random access; operations are restricted to the head of the queue .
Insertion/Deletion Complexity	Insertion: $O(\log n)$, Deletion: $O(\log n)$ (heapify operations maintain order).
Peek Complexity	Peeking (retrieving the smallest/largest element): $O(1)$.
Null Elements	Does not allow null elements.

Thread-Safety

Not synchronized; use `PriorityBlockingQueue` for thread safety.

Heap Implementation

- Internally, `PriorityQueue` uses a **binary heap**, represented as an **array**.
- In a **min-heap**, each parent node is smaller than its child nodes. For a **max-heap**, the opposite is true.

Common Operations

1. Adding an Element (`offer` or `add`)

- Inserts the element into the queue and reorders the heap.
- Complexity: $O(\log n)$ due to the heapify-up operation.

2. Removing the Head (`poll`)

- Removes the smallest/largest element (depending on the ordering).
- Complexity: $O(\log n)$ due to the heapify-down operation.

3. Peeking at the Head (`peek`)

- Retrieves the smallest/largest element without removing it.
- Complexity: $O(1)$.

Example Usage

```
import java.util.PriorityQueue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        // Default: Min-Heap
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        pq.add(10);
        pq.add(5);
```

```

        pq.add(15);
        pq.add(3);

System.out.println("Priority Queue: " + pq); // Elements are not ordered in the output

// Peek: Retrieve the smallest element
    System.out.println("Peek (Smallest): " + pq.peek()); // Output: 3

// Poll: Remove the smallest element
    System.out.println("Poll (Removed): " + pq.poll()); // Output: 3
    System.out.println("After Poll: " + pq);           // Heap reordered
}
}

```

Custom Comparator for Max-Heap

By default, PriorityQueue is a min-heap. To create a max-heap, provide a custom comparator.

```

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

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

maxHeap.add(10);
        maxHeap.add(5);
        maxHeap.add(15);
        maxHeap.add(3);

System.out.println("Max-Heap Priority Queue: " + maxHeap); // Output not ordered

// Retrieve the largest element
    System.out.println("Peek (Largest): " + maxHeap.peek()); // Output: 15
    }
}

```

When to Use

- **Priority Scheduling:** When tasks must be executed based on priority (e.g., job schedulers).
- **Dijkstra's Algorithm:** For graph traversal where the smallest weight edge is needed.
- **Event Simulation:** Handling events in order of their priority.

Key Points

- PriorityQueue does **not maintain a sorted order** for all elements but ensures the head is always the smallest/largest.
- It is efficient for **priority-based tasks** with frequent insertion and removal at the head.

Queue vs Stack

Queue vs Stack in Java

Both Queue and Stack are fundamental data structures in Java, part of the Java Collections Framework, but they are designed to work in different ways. Here's a comparison of the two:

1. Conceptual Overview

Feature	Queue	Stack
Data Structure	A collection of elements that follows FIFO (First In, First Out) .	A collection of elements that follows LIFO (Last In, First Out) .
Purpose	Used to represent a sequence of elements where the first added element is processed first .	Used to represent a sequence where the last added element is processed first .
Common Analogy	A line at a store or checkout, where the first person to join the line is the first to be served.	A stack of plates , where the plate placed on top is the first to be taken off.

2. Key Operations

Operation	Queue	Stack
Insertion	offer(E e), add(E e) - Adds an element to the rear .	push(E item) - Adds an element to the top .
Removal	poll() - Removes and returns the front element.	pop() - Removes and returns the top element.
Peek	peek() - Retrieves but does not remove the front element.	peek() - Retrieves but does not remove the top element.
Is Empty	isEmpty() - Checks if the queue is empty.	isEmpty() - Checks if the stack is empty.

3. Common Implementations

Feature	Queue	Stack
Implementations	LinkedList, PriorityQueue, ArrayDeque	Vector, Stack, ArrayDeque
Thread Safety	PriorityQueue and LinkedList are not thread-safe , but ConcurrentLinkedQueue is thread-safe.	Stack is synchronized (not ideal for high concurrency), while ArrayDeque is not thread-safe.
Performance	Operations like offer, poll have O(1) complexity.	Operations like push, pop have O(1) complexity.

4. Use Cases

Queue	Stack
Task Scheduling: Managing tasks based on priority (e.g., BFS traversal, task queues).	Undo/Redo Operations: Common in applications where the most recent operation should be undone first.

Message Handling: Used in messaging systems or buffers (e.g., producer-consumer pattern).

Expression Evaluation: Useful in evaluating postfix or prefix expressions.

Breadth-First Search (BFS): Used in graph traversal for storing nodes to visit.

Depth-First Search (DFS): Used in graph traversal for managing visited nodes or function calls.

5. Example Code: Queue vs Stack

Queue Example

```
import java.util.Queue;
import java.util.LinkedList;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();

        queue.add(10); // Adds to the rear
        queue.add(20);
        queue.add(30);

        System.out.println("Queue: " + queue);

        System.out.println("Peek: " + queue.peek()); // Retrieves but doesn't remove the front
        element
        System.out.println("Poll: " + queue.poll()); // Removes and returns the front element

        System.out.println("Queue after poll: " + queue);
    }
}
```

Stack Example

```
import java.util.Stack;

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

```

stack.push(10); // Adds to the top
stack.push(20);
stack.push(30);

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

System.out.println("Peek: " + stack.peek()); // Retrieves but doesn't remove the top
element
System.out.println("Pop: " + stack.pop()); // Removes and returns the top element

System.out.println("Stack after pop: " + stack);
}
}

```

6. Comparison Summary

Feature	Queue	Stack
Order	FIFO (First In, First Out)	LIFO (Last In, First Out)
Main Use	Handling tasks in order of arrival, scheduling, buffers.	Managing operations in reverse order, undo/redo, recursion.
Key Operations	offer, poll, peek	push, pop, peek
Thread Safety	Not thread-safe by default (use ConcurrentQueue for thread safety).	Thread-safe if using Stack, but not ideal for concurrency.

Conclusion

- **Queue** is ideal for scenarios where you need to process elements in the order they arrive (FIFO), such as in scheduling, buffering, or task management.
- **Stack** is suitable for situations requiring the reversal of operations (LIFO), like undo-redo functionality, expression evaluation, and recursion.

HashMap vs LinkedHashMap vs Hashtable vs ConcurrentHashMap vs TreeMap

Comparison of HashMap, LinkedHashMap, Hashtable, ConcurrentHashMap, and TreeMap in Java

Feature	HashMap	LinkedHashMap	Hashtable	ConcurrentHashMap	TreeMap
Ordering	No guarantee of order.	Maintains insertion order.	No guarantee of order.	No guarantee of order.	Maintains elements in sorted order (natural or comparator).
Null Keys/Values	Allows 1 null key and multiple null values.	Allows 1 null key and multiple null values.	Does not allow null keys or null values.	Does not allow null keys , but allows null values.	Does not allow null keys but allows null values.
Thread Safety	Not thread-safe.	Not thread-safe.	Thread-safe (synchronized).	Thread-safe (uses fine-grained locking).	Not thread-safe.
Performance	Faster for non-concurrent operations.	Slightly slower than HashMap due to order maintenance.	Slower than HashMap due to synchronization overhead.	Faster for concurrent operations due to segmented locks.	Slower than HashMap due to sorting overhead.
Use Case	General-purpose, non-threaded, key-value storage.	When insertion order is important.	Legacy, thread-safe map implementation.	High-performance thread-safe map for concurrent environments.	Sorted key-value storage with navigational

					operations
					.
Implementation	Backed by a hash table.	Backed by a hash table and doubly-linked list.	Backed by a hash table.	Backed by a hash table with concurrent access.	Backed by a red-black tree.
Introduced In	Java 1.2	Java 1.4	Java 1.0	Java 1.5	Java 1.2

Detailed Comparisons

1. HashMap

- **Pros:**
 - Fast and efficient for non-threaded environments.
 - Allows null keys and values.
- **Cons:**
 - No ordering of elements.
 - Not thread-safe.
- **Usage Example:**

```
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(null, "NullKey");
System.out.println(map); // {null=NullKey, 1=One}
```

2. LinkedHashMap

- **Pros:**
 - Maintains insertion order.
 - Can also maintain access order if specified (accessOrder = true).
- **Cons:**
 - Slightly slower than HashMap due to order maintenance.

- **Usage Example:**

```
LinkedHashMap<Integer, String> map = new LinkedHashMap<>();  
map.put(1, "One");  
map.put(2, "Two");  
System.out.println(map); // {1=One, 2=Two}
```

3. Hashtable

- **Pros:**

- Thread-safe for single-threaded operations.

- **Cons:**

- Legacy class; replaced by ConcurrentHashMap in most cases.
- Does not allow null keys or values.
- Lower performance due to synchronization overhead.

- **Usage Example:**

```
Hashtable<Integer, String> table = new Hashtable<>();  
table.put(1, "One");  
System.out.println(table); // {1=One}
```

4. ConcurrentHashMap

- **Pros:**

- High-performance, thread-safe implementation.
- Uses **fine-grained locking** for better concurrency (segment-based locking).

- **Cons:**

- Does not allow null keys.

- **Usage Example:**

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();  
map.put(1, "One");  
System.out.println(map); // {1=One}
```

5. TreeMap

- **Pros:**

- Maintains elements in **sorted order** (natural or custom comparator).

- Supports navigational operations (headMap, tailMap, etc.).
- **Cons:**
 - Slower than hash-based maps due to sorting overhead.
 - Does not allow null keys.
- **Usage Example:**

```

TreeMap<Integer, String> map = new TreeMap<>();
map.put(2, "Two");
map.put(1, "One");
System.out.println(map); // {1=One, 2=Two}

```

Summary Table: Key Properties

Property	HashMap	LinkedHashMap	Hashtable	ConcurrentHashMap	TreeMap
Allows Null Key	Yes (1 key)	Yes (1 key)	No	No	No
Allows Null Value	Yes	Yes	No	Yes	Yes
Order	None	Insertion Order	None	None	Sorted Order
Thread-Safe	No	No	Yes	Yes	No
Use Case	General storage	Ordered storage	Legacy, thread-safe	Concurrent environments	Sorted storage

How hashmap works

How HashMap Works in Java

HashMap in Java is a widely used data structure that stores key-value pairs. It provides efficient access to elements using a hash-based mechanism.

1. Basic Structure of HashMap

- **Bucket:** The underlying data structure of a HashMap is an array of buckets. Each bucket stores elements that hash to the same index.
- **Node:** Each bucket contains a linked list or a balanced tree of nodes. Each node stores:
 - A **key** and a **value**.
 - The **hash value** of the key.
 - A reference to the next node in the bucket (if collision occurs).

2. Hashing Mechanism

- **Hash Function:** When a key is added to the HashMap, its hash code is calculated using the `hashCode()` method of the key. The hash code is then compressed to an index value using:
$$\text{index} = \text{hash}(\text{key}) \% \text{capacity};$$
- **Index:** The index determines which bucket the key-value pair will be stored in.

3. Operations

a. Insertion

1. Calculate the hash code of the key.
2. Determine the index using the hash function.
3. Check if a node with the same key already exists in the bucket:
 - If it exists, update the value.
 - If it doesn't exist, create a new node and add it to the bucket.
4. If the bucket has too many nodes (e.g., due to collisions), it converts the bucket's linked list into a balanced tree (introduced in Java 8).

b. Retrieval

1. Calculate the hash code of the key.

2. Determine the bucket index using the hash function.
3. Search the bucket for a node with the matching key.
4. Return the value if found; otherwise, return null.

c. Collision Handling

- **Collision:** When two keys hash to the same bucket index, HashMap handles collisions using **separate chaining**:
 - Nodes are stored in a linked list (or a balanced tree if the list becomes too long).
 - All keys in a bucket are compared using the equals() method to find the correct key.

d. Rehashing

When the HashMap exceeds its **load factor** (default is 0.75), it resizes itself:

1. The capacity of the HashMap is doubled.
2. All elements are rehashed and redistributed to the new buckets.

4. Key Methods

Method	Description
put(key, value)	Adds or updates the key-value pair in the map.
get(key)	Retrieves the value associated with the given key.
remove(key)	Removes the key-value pair for the specified key.
containsKey(key)	Checks if the map contains the specified key.
containsValue(value)	Checks if the map contains the specified value.

5. Key Features

Feature	Details
Time Complexity	Average: $O(1)$ for <code>put()</code> , <code>get()</code> , and <code>remove()</code> ; Worst: $O(n)$ (if collisions occur).
Thread Safety	HashMap is not thread-safe . Use <code>ConcurrentHashMap</code> for concurrent access.
Null Handling	Allows one null key and multiple null values .
Load Factor	Default is 0.75, meaning resizing occurs when 75% of the buckets are filled.
Resizing	Doubles capacity and rehashes all keys when load factor exceeds the threshold.

6. Example Code

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        // Create a HashMap
        HashMap<String, Integer> map = new HashMap<>();

        // Add key-value pairs
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Retrieve a value
        System.out.println("Bob's age: " + map.get("Bob"));

        // Check for a key
        System.out.println("Contains 'Alice'? " + map.containsKey("Alice"));

        // Remove a key-value pair
        map.remove("Charlie");

        // Print all key-value pairs
        System.out.println("Map contents: " + map);
    }
}
```

```
}  
}
```

7. Advantages

- Fast and efficient for key-based lookups.
- Allows null keys and values.
- Automatically handles resizing.

8. Limitations

- Not thread-safe.
- Relatively slower compared to alternatives like TreeMap for sorted access.

How hashmap handles hash collision

In Java, HashMap uses **hashing** to store and retrieve key-value pairs efficiently. However, **hash collisions** can occur when multiple keys hash to the same bucket index. HashMap resolves collisions primarily using **Separate Chaining** and **Open Addressing** (though Java's HashMap implements separate chaining).

1. Separate Chaining

This is the approach used by Java's HashMap.

- **How It Works:**
 - Each bucket in the underlying array contains a linked list (or a balanced tree, starting from Java 8).
 - When a collision occurs, the new key-value pair is appended to the linked list (or inserted into the tree if it exceeds a threshold).
 - When retrieving, the equals() method is used to compare keys within the bucket.
- **Tree Conversion (Java 8+):**
 - If the number of elements in a single bucket exceeds a threshold (default: 8), the linked list is converted into a balanced binary tree (e.g., a Red-Black

Tree). This improves search time from $O(n)$ to $O(\log n)$ in heavily loaded buckets.

- **Pros:**

- Simple to implement.
- Handles high collision rates well by managing separate chains in each bucket.
- No limit on the number of elements.

- **Cons:**

- May consume additional memory for linked lists or trees.
- Performance degrades to $O(n)$ per bucket when using a linked list.

2. Open Addressing

Although not used in Java's HashMap, this is a common alternative collision resolution method in hash tables.

- **How It Works:**

- Instead of using separate chains, all entries are stored within the array itself.
- When a collision occurs, the algorithm probes other slots in the array to find an empty space to store the key-value pair.
- Common probing techniques:
 - **Linear Probing:** Look at the next slot sequentially.
 - **Quadratic Probing:** Use a quadratic function to determine the next slot.
 - **Double Hashing:** Use a secondary hash function to determine the next slot.

- **Pros:**

- Saves memory as no additional structures are required.
- Ideal for hash tables with a fixed number of elements.

- **Cons:**

- Difficult to resize; rehashing the entire table is expensive.
- Performance degrades significantly as the table fills up.

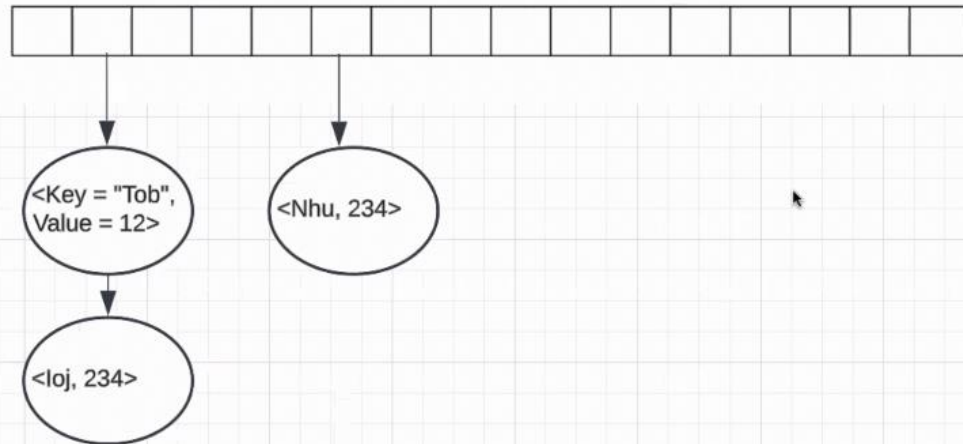
Summary of Key Differences

Feature	Separate Chaining	Open Addressing
Data Storage	Uses linked lists or trees for each bucket.	Stores all elements within the array.
Space Efficiency	Requires extra memory for chains (or trees).	More memory-efficient as no extra structures are used.
Collision Handling	Elements are added to a chain or tree.	Elements are probed for an empty slot.
Performance	$O(1)$ average, $O(n)$ worst-case (per bucket).	$O(1)$ average, performance degrades as table fills.
Usage in Java	Used by HashMap.	Not used in HashMap but in some other hash tables.

Why Java Uses Separate Chaining

- **Flexibility:** Separate chaining allows for dynamic growth, as new chains can be added without worrying about table size.
- **Performance Consistency:** Resizing is easier and collisions are handled without probing overhead.

```
HashMap<Key, Value>  
<Key = "Tob", Value = 12>  
Tob -> hashCode(33) -> % 16 = 1  
<Nhu, 234> -> hashCode(37) -> % 16 = 5  
<loj, 234> -> hashCode(17) -> %16 = 1
```



Hashtable lock

In Java, **Hashtable** is a part of the `java.util` package and is a collection class that implements a hash table data structure. It is similar to **HashMap**, but with some key differences, especially in terms of **thread safety**.

Locking in Hashtable

- **Thread Safety:**

A Hashtable is **synchronized** by default. This means that every method of a Hashtable (e.g., `get()`, `put()`, etc.) is thread-safe and can be safely used by multiple threads concurrently without additional synchronization.

- **How It Works:**

The internal synchronization is implemented by using a **single lock** for the entire Hashtable object. When one thread accesses a method of the Hashtable, it acquires a lock on the object, preventing other threads from executing any methods of that Hashtable object concurrently.

For example, when a thread calls the `put()` method on a Hashtable, it locks the entire Hashtable object to ensure that no other thread can modify the table at the same time. This lock is applied to all methods of Hashtable, even if only one method is being used.

Example of Locking in Hashtable

Here's an example of how the locking mechanism works:

```
import java.util.Hashtable;

public class HashtableExample {
    public static void main(String[] args) {
        // Create a Hashtable object
        Hashtable<String, Integer> table = new Hashtable<>();

        // Thread 1: Adding a key-value pair
        new Thread(() -> {
            synchronized (table) {
                table.put("One", 1);
                System.out.println("Thread 1 added: One=1");
            }
        }).start();

        // Thread 2: Trying to add a key-value pair while Thread 1 holds the lock
        new Thread(() -> {
            synchronized (table) {
                table.put("Two", 2);
                System.out.println("Thread 2 added: Two=2");
            }
        }).start();
    }
}
```

Locking Characteristics of Hashtable

- **Single Lock:**
All operations on a Hashtable are synchronized using a single lock. This means that only one thread can access any method of the Hashtable at any given time, even if the operations are independent of each other.
- **Performance Impact:**
Since the lock is applied to the entire object, the performance can degrade if multiple threads are trying to access the Hashtable concurrently. The synchronization can create a bottleneck, leading to increased waiting times for threads.

Comparison with HashMap

While Hashtable uses **synchronization on the entire object**, **HashMap** is **not synchronized** by default, which means multiple threads can access the HashMap concurrently without synchronization. However, if thread-safety is required, one can use **Collections.synchronizedMap()** to create a synchronized version of a HashMap.

Example:

```
Map<String, Integer> map = Collections.synchronizedMap(new HashMap<>());
```

Alternative for Thread Safety: ConcurrentHashMap

ConcurrentHashMap is a more efficient alternative to Hashtable when dealing with thread-safe operations. Unlike Hashtable, which locks the entire object for every method call, ConcurrentHashMap divides the map into segments and locks only the segment being accessed. This allows for higher concurrency.

Example:

```
ConcurrentHashMap<String, Integer> concurrentMap = new ConcurrentHashMap<>();  
concurrentMap.put("One", 1);
```

- **ConcurrentHashMap:**
 - More scalable and efficient than Hashtable in multithreaded environments.
 - Allows multiple threads to read and write concurrently by dividing the map into segments and locking them individually.
 - Does not lock the entire map, improving concurrency and reducing bottlenecks.

Summary

- **Hashtable** is thread-safe, but its synchronization uses a **single lock**, which can cause performance issues when multiple threads try to access it simultaneously.
- **ConcurrentHashMap** is a more efficient and scalable thread-safe map implementation, allowing better concurrency than Hashtable.
- **HashMap** is not thread-safe, and if thread safety is required, one can use **Collections.synchronizedMap()** to make it thread-safe.

Comparator vs comparable

In Java, **Comparator** and **Comparable** are both interfaces used for sorting objects, but they differ in their approach and usage. Below is a detailed comparison of Comparator and Comparable:

1. Comparable Interface

- **Purpose:**

The Comparable interface is used to define a natural ordering for the objects of a class. It allows the objects to be sorted in a specific order when using methods like Collections.sort() or Arrays.sort().

- **Location:**

The Comparable interface is in the java.lang package.

- **Method:**

It has one method:

int compareTo(T o);

- **Return Values:**

- A **negative integer** if the current object is less than the object o.
 - A **positive integer** if the current object is greater than the object o.
 - **Zero** if both objects are equal.

- **Usage:**

A class implements the Comparable interface to specify how its objects should be ordered.

- **Example:**

```
class Person implements Comparable<Person> {
    String name;
    int age;

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

    @Override
    public int compareTo(Person other) {
        return this.age - other.age; // Sorting by age
    }
}
```



```

    }
}

public class TestComparable {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("John", 25));
        people.add(new Person("Jane", 30));
        people.add(new Person("Tom", 22));

        Collections.sort(people); // Sorts by age (natural ordering)

        for (Person person : people) {
            System.out.println(person.name + ": " + person.age);
        }
    }
}

```

2. Comparator Interface

- **Purpose:**

The Comparator interface is used when you want to define custom sorting logic, either for a specific class or in scenarios where you want multiple ways to sort objects (e.g., sorting by name, then age).

- **Location:**

The Comparator interface is in the java.util package.

- **Method:**

It has two methods:

int compare(T o1, T o2);

boolean equals(Object obj); // Optional method, inherited from Object class

- **Return Values:**

- A **negative integer** if o1 is less than o2.
- A **positive integer** if o1 is greater than o2.
- **Zero** if both are equal.

- **Usage:**

A class does not need to implement the Comparator interface. Instead, you create a

separate Comparator class to define a custom sorting order, which can be used with sorting methods like Collections.sort() or Arrays.sort().

- **Example:**

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

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

class NameComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.name.compareTo(p2.name); // Sorting by name
    }
}

public class TestComparator {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("John", 25));
        people.add(new Person("Jane", 30));
        people.add(new Person("Tom", 22));

        Collections.sort(people, new NameComparator()); // Sorts by name

        for (Person person : people) {
            System.out.println(person.name + ": " + person.age);
        }
    }
}
```

Key Differences Between Comparator and Comparable

Feature	Comparable	Comparator
---------	------------	------------

Package	java.lang	java.util
Method(s)	compareTo(T o)	compare(T o1, T o2)
Purpose	Used for defining natural ordering of objects.	Used for defining custom ordering of objects.
Implementation Location	Class itself implements the Comparable interface.	Separate class or anonymous class implements Comparator.
Usage	Suitable when there is only one natural order.	Suitable for multiple different sorting criteria.
Flexibility	Limited to one ordering, defined within the class.	Can define multiple sorting strategies for the same class.
Inheritance	Requires modification of the class itself to implement.	Does not require modifying the class, just creating a comparator.
Example	Sorting by a natural field, like age.	Sorting by custom logic, like name or date of birth.

When to Use Which?

- **Use Comparable:**
 - When you want to define a **natural order** for objects.
 - When there's only one sorting criterion (e.g., sorting employees by their salary or age).
 - When you can modify the class itself.
- **Use Comparator:**
 - When you want to define **multiple sorting orders** or criteria.
 - When you cannot modify the class directly, or you want to sort using fields not present in the natural order.
 - When sorting by different attributes (e.g., sorting by name, then by age).

Additional Notes

- **Java 8+:** Java 8 introduced default and static methods in `Comparator`, allowing for more concise code. You can now use `Comparator` methods like `comparing()`, `thenComparing()`, and `reversed()` to create comparators without explicitly implementing the `compare()` method.

Example:

```
List<Person> people = new ArrayList<>();
people.add(new Person("John", 25));
people.add(new Person("Jane", 30));
people.add(new Person("Tom", 22));
```

```
// Sorting by age using Comparator's static method
people.sort(Comparator.comparingInt(Person::getAge));
```

```
// Sorting by name then by age
people.sort(Comparator.comparing(Person::getName).thenComparingInt(Person::getAge));
```