**🔸 What is Time Complexity?**

**Time Complexity** is the amount of **time taken** by an algorithm or operation to run **as a function of the input size** (usually denoted as **n**).

👉 In collections, it reflects how long operations like add(), get(), remove(), or contains() will take.

| **Big-O Notation** | **Meaning** | **Example** |
| --- | --- | --- |
| O(1) | Constant time | HashMap.get(key) |
| O(log n) | Logarithmic time | TreeSet.add(value) |
| O(n) | Linear time | ArrayList.contains(value) |
| O(n log n) | Quasi-linear time | Sorting operations |
| O(n²) | Quadratic time | Nested loops |

**🔹 What is Space Complexity?**

**Space Complexity** is the **amount of memory** (RAM) used by the algorithm/data structure **relative to the input size**.

In Java Collections, it includes:

* Memory for actual elements
* Overhead due to internal structure (like array resizing, pointers, hash tables)
* **Time & Space Complexities of Common Java Collections**

| **Collection Type** | **Operation** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- | --- |
| **ArrayList** | Add at end | O(1) (amortized) | O(n) |
|  | Add at index | O(n) |  |
|  | Get(index) | O(1) |  |
|  | Contains | O(n) |  |
|  | Remove(index) | O(n) |  |
| **LinkedList** | Add at head/tail | O(1) | O(n) |
|  | Get(index) | O(n) |  |
|  | Remove(index) | O(n) |  |
| **HashSet** | Add/Remove/Check | O(1) (avg), O(n) worst | O(n) |
| **TreeSet** | Add/Remove/Check | O(log n) | O(n) |
| **HashMap** | Get/Put | O(1) (avg), O(n) worst | O(n) |
| **TreeMap** | Get/Put | O(log n) | O(n) |
| **PriorityQueue** | Add/Remove | O(log n) | O(n) |

* **Real Example: ArrayList vs LinkedList**
* ArrayList uses a dynamic array.
  + Fast random access: get(index) → O(1)
  + Slow insertions in middle: add(index) → O(n)
* LinkedList uses doubly-linked nodes.
  + Slower access: get(index) → O(n)
  + Faster insert/remove at head/tail → O(1)
* Choose based on **what operations are frequent** and **performance needs**.

**1. Banking Use Case: Transaction History**

**Scenario:**

A banking app needs to maintain **transaction history** for users:

* Users often request the **last few transactions** (e.g., last 5 or 10).
* **Insertion** is frequent (every transaction).
* **Searching or removing** is rare.

**✅ Best Collection: LinkedList**

**Reason:**

* addFirst() or addLast() → O(1)
* get(index) → O(n), but acceptable if only recent transactions are needed
* Memory use is higher (extra pointers), but manageable.

**Time & Space Complexity:**

| **Operation** | **Complexity** |
| --- | --- |
| Insert at end | O(1) |
| Get by index | O(n) |
| Space | O(n) |

🟢 Use LinkedList if you need fast insertions and don’t care much about random access.

**2. Telecom Use Case: Call Record Lookup**

**Scenario:**

A telecom provider logs all **call records** and allows users to **search** for records based on phone numbers.

**✅ Best Collection: HashMap<String, CallRecord>**

**Reason:**

* Key is the **phone number** (String).
* Fast lookup, add, and delete operations → O(1) average time.
* Call logs can grow large, so **efficient access is critical**.

**Time & Space Complexity:**

| **Operation** | **Complexity** |
| --- | --- |
| Put/get | O(1) avg |
| Remove | O(1) avg |
| Space | O(n) |

🟢 HashMap is highly efficient for key-based lookup, ideal for massive telecom datasets.

**String Manipulation in Java**

String manipulation is a **core part of Java programming**, especially when working with user input, files, APIs, or databases.

**1. Basic String Operations**

String s = "Hello Java";

System.out.println(s.length()); // 10

System.out.println(s.toUpperCase()); // "HELLO JAVA"

System.out.println(s.toLowerCase()); // "hello java"

System.out.println(s.charAt(1)); // 'e'

System.out.println(s.substring(0, 5)); // "Hello"

System.out.println(s.contains("Java"));// true

System.out.println(s.indexOf("a")); // 7

**2. String Concatenation**

String a = "Hello";

String b = "World";

String c = a + " " + b; // "Hello World"

⚠️ **Note**: Using + in loops is inefficient due to immutability. Prefer StringBuilder.

**3. Using StringBuilder (Mutable & Efficient)**

StringBuilder sb = new StringBuilder("Hello");

sb.append(" Java");

sb.insert(5, " World");

sb.replace(0, 5, "Hi");

System.out.println(sb.toString()); // "Hi World Java"

Use StringBuilder when **modifying strings frequently**, especially in loops.

**4. Splitting and Joining Strings**

String data = "apple,banana,mango";

String[] fruits = data.split(",");

for (String fruit : fruits) {

System.out.println(fruit);

}

String joined = String.join(" | ", fruits);

System.out.println(joined); // "apple | banana | mango"

**5. Trimming and Replacing**

String raw = " Hello Java ";

System.out.println(raw.trim()); // "Hello Java"

String msg = "I like Java";

System.out.println(msg.replace("Java", "Python")); // "I like Python"

**6. Reversing a String**

String input = "Java";

StringBuilder sb = new StringBuilder(input);

System.out.println(sb.reverse()); // "avaJ"

**7. String Comparison**

String x = "Hello";

String y = "hello";

System.out.println(x.equals(y)); // false

System.out.println(x.equalsIgnoreCase(y));// true

**8. Convert Between String and Other Types**

int num = 123;

String strNum = String.valueOf(num); // int to String

int parsed = Integer.parseInt(strNum); // String to int

**9. Regular Expressions (Regex)**

String email = "user@example.com";

boolean isValid = email.matches("\\w+@\\w+\\.com");

System.out.println(isValid); // true

**10. Useful String Utility Methods**

| **Method** | **Description** |
| --- | --- |
| startsWith("prefix") | Checks prefix |
| endsWith("suffix") | Checks suffix |
| toCharArray() | Converts string to character array |
| replaceAll(regex, value) | Replaces using regex |
| split(delimiter) | Splits string |
| String.format() | Formatted string creation |

**Real-time Example: Masking Credit Card Number**

public class MaskCard {

public static void main(String[] args) {

String cardNumber = "1234567812345678";

String masked = "XXXX-XXXX-XXXX-" + cardNumber.substring(12);

System.out.println(masked); // XXXX-XXXX-XXXX-5678

}

}

**What is an Iterator in Java Collection?**

An **Iterator** in Java is an object that enables you to **traverse (iterate) over elements** in a collection **one at a time**, without exposing the internal structure of the collection.

**Why Use Iterator?**

* To **safely traverse** a collection (List, Set, etc.)
* To **remove elements** during iteration without ConcurrentModificationException
* Works with **Legacy classes** (e.g., Vector, Hashtable)

**Basic Iterator Interface Methods**

Iterator<E> iterator();

| **Method** | **Description** |
| --- | --- |
| hasNext() | Returns true if there is another element |
| next() | Returns the next element |
| remove() | Removes the current element (optional op) |

**🔸 Example: Iterating through a List**

import java.util.\*;

public class IteratorExample {

public static void main(String[] args) {

List<String> names = new ArrayList<>();

names.add("Amit");

names.add("Preety");

names.add("Rahul");

Iterator<String> it = names.iterator();

while (it.hasNext()) {

String name = it.next();

System.out.println(name);

}

}

}

**🔸 Removing Elements While Iterating**

Iterator<String> it = names.iterator();

while (it.hasNext()) {

String name = it.next();

if (name.equals("Preety")) {

it.remove(); // Safe removal

}

}

⚠️ Never use names.remove("Preety") directly during iteration — it will throw ConcurrentModificationException.

**🔸 Iterator vs Enhanced for-loop**

for (String name : names) {

System.out.println(name);

}

✅ Shorter  
❌ Cannot remove elements

**🔸 Iterator for Other Collections**

* Set – You must use Iterator to loop (no indexing)
* Map – You can iterate keys/values using entrySet():

Map<Integer, String> map = new HashMap<>();

map.put(1, "A");

map.put(2, "B");

for (Map.Entry<Integer, String> entry : map.entrySet()) {

System.out.println(entry.getKey() + " -> " + entry.getValue());

}

Or using Iterator:

Iterator<Map.Entry<Integer, String>> it = map.entrySet().iterator();

while (it.hasNext()) {

Map.Entry<Integer, String> entry = it.next();

System.out.println(entry.getKey() + " => " + entry.getValue());

}

**Types of Iterators in Java**

| **Type** | **Description** |
| --- | --- |
| **Iterator** | Basic read-remove support (List, Set) |
| **ListIterator** | Only for List. Supports **both directions** and add(), set() |
| **Enumeration** | Legacy interface (used in Vector, etc.) |
| **Spliterator** | For **parallel traversal** (Java 8+) |

**Real-World Use Case:**

In a **banking application**, if a list of flagged transactions is under review and we want to **remove suspicious ones while scanning**, an Iterator allows safe removal without interrupting the iteration process.

**When to Use Iterator Instead of Loops**

| **Situation** | **Why Iterator is Better** |
| --- | --- |
| 🔄 **No Index Access (e.g., Set, Queue)** | Collections like HashSet, TreeSet, or Queue **don’t support indexing**, so **you can’t use a traditional for loop**. You **must** use an Iterator or enhanced for-each. |
| 🗑️ **Need to Remove Elements During Iteration** | Using a for-each or classic for loop and modifying the collection (like list.remove()) causes **ConcurrentModificationException**. With an Iterator, you can safely call iterator.remove(). |
| 🧩 **Uniform Way to Traverse Any Collection** | If you're writing **generic code** (e.g., utility method), using Iterator lets you **treat all collections the same**, regardless of their specific type. |
| 📉 **Improves Readability in Some Complex Scenarios** | When you’re doing **conditional removals, skips, or nested logic**, iterators can sometimes provide cleaner control than index-based loops. |

**When NOT to Use Iterator (Use Loops Instead)**

| **Situation** | **Better Option** |
| --- | --- |
| 📍 You need **index-based access** | Use a for loop for List when you need list.get(i) or modify based on index. |
| ⏱️ Performance critical & simple read | For small, predictable collections, a for-each or stream may be more concise and readable. |
| 🛠️ You’re using **Streams/Lambdas** (Java 8+) | Use stream().filter().forEach() when readability and functional programming are priorities. |

**Real Examples**

**✔️ Use Iterator – Removing elements from a Set**

Set<String> set = new HashSet<>();

set.add("India");

set.add("China");

set.add("USA");

Iterator<String> it = set.iterator();

while (it.hasNext()) {

if (it.next().equals("China")) {

it.remove(); // Safe removal

}

}

**❌ Don’t Use Iterator – Simple Indexed Access in a List**

List<String> list = Arrays.asList("A", "B", "C");

for (int i = 0; i < list.size(); i++) {

System.out.println(list.get(i)); // Indexed access

}

**Summary: Rule of Thumb**

| **Use Case** | **Best Approach** |
| --- | --- |
| Read-only iteration | Enhanced for-each loop |
| Modify collection during iteration | Iterator (with remove()) |
| Need index control (List only) | Traditional for loop |
| Use Streams or filters | Java 8+ Stream API |

**🔷 What is a Graph in Data Structures?**

A **Graph** is a **non-linear data structure** that consists of:

* **Vertices (nodes)** — points/entities in the graph.
* **Edges (links)** — connections between pairs of vertices.

Graphs are used to model **networks** like:

* Social media connections
* Maps (GPS)
* Computer networks
* Web page links
* Airline routes
* Project task dependencies

**Components of a Graph**

Graph G = (V, E)

V = {set of vertices}

E = {set of edges between vertices}

For example:

* **Vertices**: Cities = {Delhi, Mumbai, Kolkata}
* **Edges**: Roads = {(Delhi, Mumbai), (Mumbai, Kolkata)}

**Types of Graphs**

**1. Directed vs Undirected**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Undirected** | Edges have **no direction** | Facebook friendship |
| **Directed (Digraph)** | Edges have **direction** (A → B) | Twitter following |

**2. Weighted vs Unweighted**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Unweighted** | All edges have equal cost | Friend connection |
| **Weighted** | Each edge has a cost/weight | Road distance, latency |

**3. Cyclic vs Acyclic**

| **Type** | **Description** |
| --- | --- |
| **Cyclic** | At least one cycle (a path that loops back) |
| **Acyclic** | No cycles |

Special Case:

* **DAG (Directed Acyclic Graph)** → Used in scheduling, compiler optimization, version control

**4. Connected vs Disconnected**

| **Type** | **Description** |
| --- | --- |
| **Connected** | Every vertex can be reached from any other |
| **Disconnected** | Some vertices are not connected |

**Representation of Graphs**

**1. Adjacency Matrix**

A 2D array (V x V) where:

* 1 means an edge exists between nodes.
* For weighted graphs, use weight instead of 1.

// Example: Graph with 3 nodes (0,1,2)

int[][] matrix = {

{0, 1, 0},

{1, 0, 1},

{0, 1, 0}

};

**Pros**: Fast lookup  
**Cons**: Space O(V²), even for sparse graphs

**2. Adjacency List**

A Map or array of Lists where each index holds a list of adjacent nodes.

Map<Integer, List<Integer>> graph = new HashMap<>();

graph.put(0, Arrays.asList(1));

graph.put(1, Arrays.asList(0, 2));

graph.put(2, Arrays.asList(1));

**Pros**: Space efficient  
**Cons**: Slower lookup than matrix (O(V) instead of O(1))

**Graph Traversal Algorithms**

**1. DFS (Depth-First Search)**

* Explore as far as possible along a branch before backtracking.
* Uses a **stack** or **recursion**.

void dfs(int node, boolean[] visited, List<List<Integer>> adj) {

visited[node] = true;

for (int neighbor : adj.get(node)) {

if (!visited[neighbor]) {

dfs(neighbor, visited, adj);

}

}

}

**2. BFS (Breadth-First Search)**

* Explore all neighbors at current depth before going deeper.
* Uses a **queue**.

void bfs(int start, List<List<Integer>> adj) {

boolean[] visited = new boolean[adj.size()];

Queue<Integer> q = new LinkedList<>();

visited[start] = true;

q.add(start);

while (!q.isEmpty()) {

int node = q.poll();

for (int neighbor : adj.get(node)) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.add(neighbor);

}

}

}

}

**Real-World Use Cases of Graphs**

| **Domain** | **Use Case** |
| --- | --- |
| **Social Media** | Finding mutual friends, shortest connection |
| **Navigation** | Google Maps → Shortest path (Dijkstra’s algorithm) |
| **Telecom** | Network routing, packet delivery |
| **Banking** | Fraud detection (loop detection in transactions) |
| **Web** | Search engines (PageRank algorithm) |
| **AI/ML** | Neural networks (as DAGs) |

**Common Graph Problems**

* Shortest path (Dijkstra, Bellman-Ford)
* Cycle detection (DFS + visited stack)
* Connected components (DFS or BFS)
* Topological sort (for DAGs)
* Minimum spanning tree (Prim, Kruskal)
* Bipartite check

**Collection Utility Classes in Java**

Java provides **utility classes** in the java.util package to **manipulate, search, sort, and synchronize** collections. These are **helper classes** that simplify common operations on collections like List, Set, Map, etc.

**1. Collections Class (java.util.Collections)**

A utility class with **static methods** to operate on or return collections.

**🔧 Common Methods:**

| **Method** | **Description** |
| --- | --- |
| Collections.sort(List) | Sorts the list in natural order |
| Collections.sort(List, Comparator) | Sorts using a custom comparator |
| Collections.reverse(List) | Reverses the elements in a list |
| Collections.shuffle(List) | Randomly shuffles the list |
| Collections.max(Collection) | Returns the maximum element |
| Collections.min(Collection) | Returns the minimum element |
| Collections.frequency(Collection, Object) | Counts occurrences |
| Collections.binarySearch(List, key) | Searches sorted list using binary search |
| Collections.synchronizedList(List) | Returns thread-safe list wrapper |
| Collections.unmodifiableList(List) | Returns a read-only list wrapper |

**Example:**

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

List<String> names = new ArrayList<>(Arrays.asList("Preety", "Amit", "Rahul"));

Collections.sort(names); // Sort

Collections.reverse(names); // Reverse

Collections.shuffle(names); // Random order

System.out.println(Collections.max(names)); // Lexicographically max

}

}

**2. Arrays Class (java.util.Arrays)**

Mainly used for working with **arrays**, not collections, but often used with lists.

**Common Methods:**

| **Method** | **Description** |
| --- | --- |
| Arrays.asList(T...) | Converts array to List (fixed-size) |
| Arrays.sort(array) | Sorts the array |
| Arrays.binarySearch(array, key) | Binary search on sorted array |
| Arrays.equals(arr1, arr2) | Compares two arrays |
| Arrays.copyOf(array, newLength) | Copies array to new length |
| Arrays.toString(array) | Converts array to readable string |

**Example:**

String[] cities = {"Delhi", "Mumbai", "Bangalore"};

List<String> cityList = Arrays.asList(cities);

Collections.sort(cityList);

System.out.println(cityList);

**3. Objects Class (java.util.Objects)**

Utility methods for working with objects (null-safe), used frequently in custom classes, especially in **collections with custom sorting or hashing**.

| **Method** | **Description** |
| --- | --- |
| Objects.equals(a, b) | Null-safe equality check |
| Objects.hashCode(obj) | Null-safe hashCode |
| Objects.requireNonNull(obj) | Throws NPE if object is null |
| Objects.compare(a, b, comp) | Null-safe comparison with Comparator |

**Use Case Example: Sorting with Comparator and Collections.sort()**

class Employee {

String name;

int salary;

Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

public String toString() {

return name + " - " + salary;

}

}

public class SortExample {

public static void main(String[] args) {

List<Employee> employees = new ArrayList<>();

employees.add(new Employee("Amit", 40000));

employees.add(new Employee("Preety", 50000));

Collections.sort(employees, (e1, e2) -> e2.salary - e1.salary); // Descending by salary

System.out.println(employees);

}

}

**Summary of Collection Utility Classes**

| **Class** | **Purpose** |
| --- | --- |
| Collections | Utility for Collection objects (List, Set, etc.) |
| Arrays | Utility for working with arrays |
| Objects | Utility for working with Object types (null-safe) |