**Iterable (Interface)**

**|**

**-----------------------------------------------------------------------**

**| |**

**Collection (i) Map (Interface)**

**|**

**------------------------------------------------**

**| | |**

**List Set Queue**

**| | |**

**ArrayList HashSet PriorityQueue**

**LinkedList LinkedHashSet ArrayDeque**

**Vector TreeSet**

**Stack**

**|**

**Map (Interface)**

**├─ HashMap → LinkedHashMap**

**├─ SortedMap → TreeMap**

**└─ ConcurrentMap → ConcurrentHashMap, ConcurrentSkipListMap**

**Iterable**

Iterable is the root interface in the Java Collection hierarchy.

The Iterable interface represents a collection of elements that can be iterated (looped) one by one, usually using an iterator or enhanced for-loop (for-each loop).

**Is Map part of Collection?**

No, Map is not a child interface of Collection.

But it is part of the Java Collections Framework (JCF).

**What is the Java Collections Framework?**

A set of interfaces, classes, and algorithms for storing, retrieving, manipulating, (and iterating over) groups of objects.

Provides ready-made data structures (like lists, sets, queues, and maps) and supporting algorithms (such as sorting and searching) .

Helps reduce boilerplate code, improve performance, and make implementations interchangeable

**LIST**

In Java, the List is an interface within the Java Collections Framework, It represents an ordered collection of elements, meaning that elements are stored and retrieved based on their insertion order and can be accessed by their numerical position (index).

* Maintains insertion order.
* Allows duplicate values.
* Supports null elements .

**Core Implementations of List**

**ArrayList**

* Structure: Dynamic, resizable array.
* Fast random access: get(index) is O(1).
* Commonly used: Best for frequent retreving data

**LinkedList**

* **Structure**: Doubly-linked list.
* Fast insert/remove at ends or middle: **O(1)** (once positioned).
* **Use case**: Ideal when frequent insertions/deletions are needed.

**SET**

In Java, the Set interface (from java.util) represents an unordered collection of unique elements. Key characteristics include:

* No duplicate values — any attempt to add a duplicate is ignored.
* Doesn't guarantee order unless using a specific implementation.
* Supports a single null element (depending on implementation)

**Core Implementations of Set**

**HashSet**

* Structure: Backed by a hash table (via HashMap).
* Ordering: No guaranteed order—iteration order is unpredictable
* Allows one null
* Use Case: Use it when you need quick lookups and don't care about ordering.

**LinkedHashSet**

* Structure: Hash table plus a doubly-linked list to maintain order
* Ordering: Maintains insertion order
* Allows one null
* Use Case: Ideal when preserving insertion order is

**TreeSet**

* Structure: Balanced Red-Black Tree (backed by TreeMap), implementing NavigableSet
* Ordering: Keeps elements in sorted order, either natural or via a Comparator
* Null Support: Does not allow null (throws NullPointerException)
* Use Case: Use it when you need a sorted set and in-order operations.

**MAP**

Map<K, V> (in java.util) represents an association between unique keys and corresponding values. It is part of the Java Collections Framework but does not extend the Collection interface. Ideal for use cases like lookup tables, caching, grouping data, and any scenario requiring fast access by key .

No duplicate keys – each key is unique; adding a duplicate replaces the existing value

Values may be duplicated, and some implementations allow null keys and/or values

 Provides views over its contents:

* keySet() → a Set of keys
* values() → a Collection of values
* entrySet() → a Set of key–value pairs (Map.Entry<K,V>)

**Core Implementations of MAP**

**HashMap<K,V>**

* Structure: Hash table
* Order: No guaranteed order;
* allows 1 null key and multiple null values
* Use case: General-purpose, fast access when order doesn’t matter.

**LinkedHashMap<K,V>**

* Structure: Hash table + linked list
* Order: Maintains insertion order (or access order if configured)
* Use case: When consistent iteration order matters (e.g., caching, LRU).

**TreeMap<K,V>**

* Structure: Red-Black tree
* Order: Sorted by natural order or custom Comparator;
* does not allow null keys [prepinsta.com+11geeksforgeeks.org+11codingshuttle.com+11](https://www.geeksforgeeks.org/map-interface-java-examples/?utm_source=chatgpt.com).
* Use case: When a sorted map is needed (range queries, ordering).

**List Methods**

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

list.add("A");

list.add("B");

list.add(1, "C"); // ["A", "C", "B"]

String x = list.get(1); // "C"

int i = list.indexOf("B"); // 2

list.remove("C"); // ["A", "B"]

list.replaceAll(s -> s.toLowerCase()); // ["a","b"]

list.sort(Comparator.naturalOrder()); // ["a","b"]

**Set Methods**

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

set.add("A");

set.add("B");

set.add("A"); // duplicate ignored

System.out.println(set.size()); // 2

set.remove("B");

List<String> other = List.of("A", "C");

set.addAll(other); // union: [A, C]

set.retainAll(List.of("A")); // intersection: [A]

set.removeAll(List.of("A")); // difference: []

**MAP Methods**

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

map.put("Alice", 30);

map.putIfAbsent("Bob", 25); // Bob → 25

map.getOrDefault("Charlie", 0); // returns 0

map.replace("Bob", 25, 26); // Bob → 26

map.computeIfAbsent("Dave", k -> 100); // Dave → 100

map.merge("Alice", 5, Integer::sum); // Alice → 35

map.remove("Bob", 26); // removes Bob

map.compute("Alice", (k, v) -> v != null ? v + 1 : null); // Alice → 36

map.replaceAll((k, v) -> v \* 2); // double all values

map.forEach((k, v) -> System.out.println(k + " = " + v));

**Real-time automation testing Usage**

**List – Ordered Collection (Allows Duplicates)**

**Storing Web Elements**

List<WebElement> allLinks = driver.findElements(By.tagName("a"));

for (WebElement link : allLinks) {

System.out.println(link.getText());

}

**Test Data from Excel or CSV**

List<String> testData = readExcelData(); // "username", "password", etc.

**Dropdown Values Validation**

List<WebElement> options = dropdown.getOptions();

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

for (WebElement opt : options) {

actualValues.add(opt.getText());

}

**Set – Unique Elements (No Duplicates)**

**Real-time Use Cases:**

**Validating Uniqueness of Dropdown Options**

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

for (WebElement opt : dropdownOptions) {

uniqueOptions.add(opt.getText());

}

if (uniqueOptions.size() == dropdownOptions.size()) {

System.out.println("All options are unique.");

}

**Storing Unique Test Data**

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

userIds.add("user1");

userIds.add("user2");

**Removing Duplicates from a List**

List<String> names = Arrays.asList("John", "Alice", "John");

Set<String> uniqueNames = new HashSet<>(names);

**Map – Key-Value Pairs**

**Real-time Use Cases:**

**Storing Test Data with Keys**

Map<String, String> userData = new HashMap<>();

userData.put("username", "ajayreddy");

userData.put("password", "pass123");

**Reading JSON Responses in API Testing**

Response res = given().get("/users");

Map<String, Object> json = res.jsonPath().getMap("");

System.out.println(json.get("name")); // e.g., returns "Ajay"

**Cucumber DataTables (Step Definitions)**

@Given("user enters credentials")

public void enterCredentials(Map<String, String> data) {

loginPage.enterUsername(data.get("username"));

loginPage.enterPassword(data.get("password"));

}

**Locator Storage**

Map<String, By> locators = new HashMap<>();

locators.put("loginButton", By.id("login"));

driver.findElement(locators.get("loginButton")).click();