# Indirection

means referring to something **not directly**, but **through an intermediate reference**.

In Java, indirection often means **using an interface or abstract class** to refer to an object, instead of directly using the class name.

#### **Why use Indirection?**

* **Flexibility** You can change the actual class later without changing the rest of the code
* **Loose Coupling** Code depends on a general contract (interface), not a specific implementation.
* **Easy Maintenance** If you want to change the implementation (e.g., from array-based to linked list), you only change one line.
* **Supports Polymorphism** You can treat different objects in the same way if they share a common interface.

**Indirection = Flexibility**

Instead of binding to a specific thing early, you use an abstract reference and decide the concrete thing later.

used heavily in **Java Collections Framework.**

| **Queue<String> q = new ArrayQueue<>(); // indirect reference  // Later we can switch to: q = new ListQueue<>();** |
| --- |

Using the Queue interface to declare the object — this is indirection. The actual object (ArrayQueue or ListQueue) can change.

# Collections

* Collections are data structures for storing groups of objects.
* Java organizes collections using interfaces and abstract classes for flexibility.
* Key Interfaces: Collection, List, Set, etc.
* Key Classes: ArrayList, HashSet, etc.
* Key Distinction: Collections (like lists, sets) are for groups of values; maps (like dictionaries) are for key-value pairs.

### **Main Methods:**

* add(E e): Adds an element.
* clear(): Removes all elements
* iterator(): Returns an iterator to traverse the collection.
* size(): Returns the number of elements.
* contains(Object o): Checks if an element is present.
* remove(Object o): Removes an element if present.
* isEmpty(): Checks if empty.
* addAll(Collection<? extends E> c): Adds all elements from another collection.
* removeAll(Collection<?> c): Removes all elements present in another collection.
* containsAll(Collection<?> c): Checks if all elements of a collection are present.

# Iterators

Traverse a collection without exposing its internal structure.

Methods:

* hasNext(): Checks if there are more elements.
* next(): Returns the next element.
* remove(): Removes the last element returned by next().

| Collection<String> cstr = new ArrayList<>(); cstr.add("apple"); cstr.add("banana"); Iterator<String> iter = cstr.iterator(); while (iter.hasNext()) {  String element = iter.next();  System.out.println(element);  // To remove "apple" if it matches a condition  if (element.equals("apple")) {  iter.remove();  } } |
| --- |

# For-Each Loop

Internally uses an iterator.

| Collection<String> cstr = new ArrayList<>(); cstr.add("apple"); cstr.add("banana"); for (String element : cstr) {  System.out.println(element); } |
| --- |

# AbstractCollection

# The AbstractCollection class in Java is an abstract (skeletal) implementation of the Collection interface. Its main purpose is to minimize the effort required to create new collection classes by providing default implementations for most of the methods defined in the Collection interface.

AbstractCollection provides implementations for common methods such as contains, toArray, isEmpty, remove, containsAll, addAll, removeAll, retainAll, and clear

Instead of implementing all Collection methods from scratch, you can extend AbstractCollection and only implement what is necessary for your custom collection.

## List Interface

* Extends Collection: Represents ordered collections (elements have a sequence: first, second, etc.).
* Key Features:
  + Random Access: Can get/set element at index (e.g., get(i), set(i, e)).
  + List Iterator: More powerful than basic iterator—can go forwards and backwards, and insert/remove at current position.
* Methods: add(int index, E e), remove(int index), get(int index), set(int index, E e).
* Random Access vs. Sequential Access:
  + ArrayList: Supports efficient random access (constant time for get(i)).
  + LinkedList: Sequential access (linear time for get(i)), but efficient for insert/delete in the middle.

| List<String> list = new ArrayList<>(); list.add("A"); list.add(1, "B"); // Insert at index 1 String s = list.get(1); // "B" |
| --- |

## Set Interface

* Extends Collection: Represents unordered collections with no duplicates.
* Key Features:
  + No Duplicates: add(E e) returns false if element already exists.
  + No Order Guarantee: Elements can be iterated, but order is not defined.
* Implementations:
  + HashSet: Uses hash table for fast membership checks (average O(1) for add/contains).
  + TreeSet: Uses balanced tree for sorted order (logarithmic time for add/contains).

| Set<String> set = new HashSet<>(); set.add("A"); set.add("A"); // Returns false, no duplicate added |
| --- |

## Queue Interface

* Extends Collection: Represents ordered collections with restricted access (insert at rear, remove from front).
* Key Features:
  + Basic Operations: add(E e), remove() (throws exception if empty), offer(E e) (returns false if full), poll() (returns null if empty), peek() (returns head without removing).
* Double-Ended Queue (Deque): Insert/remove at both ends (addFirst, addLast, removeFirst, removeLast).
* PriorityQueue: Elements removed based on priority, not insertion order.

| Queue<String> queue = new PriorityQueue<>(); queue.add("B"); queue.add("A"); String head = queue.poll(); // "A" |
| --- |

# Iterator Usage

* + Basic Iterator: Only forward traversal.
  + List Iterator: Forward and backward traversal, insert/remove at current position.

| ListIterator<String> it = list.listIterator(); while (it.hasNext()) {  String s = it.next();  if (s.equals("B")) it.add("C"); // Insert after "B" } |
| --- |

## Vector in Java

* Definition:
  + Vector is a dynamic, array-like data structure that can grow or shrink as needed.
  + It implements the List interface and is part of the Java Collections Framework.
* Key Features:
  + Ordered: Maintains insertion order.
  + Indexed: Supports zero-based index access.
  + Duplicates Allowed: Permits multiple occurrences of the same element.
  + Synchronized: All methods are thread-safe, making it suitable for concurrent environments, but slower than unsynchronized alternatives like ArrayList.
  + Dynamic Resizing: Automatically increases capacity as elements are added. Capacity can be specified at creation, and an optional increment can be set.

| Vector<String> names = new Vector<>(); names.add("Alice"); names.add("Bob"); String name = names.get(1); // "Bob" |
| --- |

## Stack

* The Stack class represents a last-in, first-out (LIFO) stack of objects.
* It extends the Vector class, inheriting all its methods but adding five stack-specific operations: push, pop, peek, empty, and search.

Key Methods:

* push(E item): Adds an item to the top of the stack.
* pop(): Removes and returns the top item.
* peek(): Returns (but does not remove) the top item.
* empty(): Tests if the stack is empty.
* search(Object o): Returns the 1-based position from the top where the object is located, or -1 if not found

| Stack<String> stack = new Stack<>(); stack.push("A"); stack.push("B"); String top = stack.peek(); // "B" String popped = stack.pop(); // "B" |
| --- |

## Priority Queue

* + A priority queue is a special type of queue where each element has a priority.
  + Elements are retrieved in order of their priority, not by insertion order (FIFO).
  + Java provides the java.util.PriorityQueue class for this purpose.
  + Ordering:
    - Elements are ordered by their natural ordering (ascending by default) or by a custom Comparator at creation.
    - The head of the queue is always the least element according to the specified ordering.

//refer to codes shared for Priority Queue

# Map Interface

* Map in Java is the interface for key-value stores (like Python dictionaries).
* Collection interface is for single-value collections (lists, sets, etc.), while Map is for key-value pairs and is separate from Collection.
* Map is generic: Map<K, V> where K is key type and V is value type.

| **Operation** | **Description** | **Example (Java)** |
| --- | --- | --- |
| put(K key, V value) | Inserts or updates a key-value pair, returns previous value (or null) | map.put("Alice", 10); |
| get(Object key) | Returns value for key, or null if not present | map.get("Alice"); |
| containsKey(Object key) | Checks if key exists | map.containsKey("Alice"); |
| containsValue(Object value) | Checks if value exists | map.containsValue(10); |

getOrDefault: Returns value or default if key not present.

| int current = map.getOrDefault("word", 0); map.put("word", current + 1); |
| --- |

putIfAbsent: Inserts only if key is not present.

| map.putIfAbsent("player", 0); map.put("player", map.get("player") + newScore); |
| --- |

## Extracting Keys, Values, and Entries

* keySet(): Returns a Set<K> of keys.

| Map<String, Integer> map = new HashMap<>(); map.put("Apple", 3); map.put("Banana", 2);  // Iterate over keys for (String key : map.keySet()) {  Integer value = map.get(key);  System.out.println(key + " => " + value); } |
| --- |

* values(): Returns a Collection<V> of values.
* entrySet(): Returns a Set<Map.Entry<K, V>> of key-value pairs.

| for (Map.Entry<String, Integer> entry : map.entrySet()) {  String key = entry.getKey();  Integer value = entry.getValue();  // Do something with key and value } |
| --- |

## Map Implementations

| **Implementation** | **Ordering** | **Underlying Structure** | **Use Case** |
| --- | --- | --- | --- |
| HashMap | Unordered | Hash table | General purpose |
| TreeMap | Sorted by key | Balanced search tree | Sorted keys needed |
| LinkedHashMap | Insertion/access | Hash table + linked list | Maintain insertion/access order |

* Map.Entry is a **nested interface** inside the Map interface.
* It **represents a single key-value pair** in a Map.

**Example:** When you call map.entrySet(), you get a Set of Map.Entry<K, V> — each entry has:

* getKey()
* getValue()
* setValue(V value) (optional

So instead of looping over keys & doing map.get(key), you use Map.Entry to work directly with **pairs**.