T.I.M.E.
Triumphant Institute of Management Education Pvt.Ltd.

Advance Educational Activities Pvt. Ltd.

Unit 5: Exceptional Handling & File Handling

5.1.1. Basics of Multithreading

Analogy:

Imagine you're cooking while listening to music and downloading a file — all happening simultaneously. That's **multitasking**, and in Java, it's called **multithreading**.

What is Multithreading?

- **Multithreading** is the ability of a program to execute **multiple threads concurrently**.
- A **thread** is a **lightweight subprocess**—the smallest unit of processing.
- Java supports multithreading via the **java.lang.Thread** class and the **Runnable** interface.

Why Use Multithreading?

BENEFIT	EXPLANATION
BETTER CPU UTILIZATION	Makes full use of processor cores
FASTER EXECUTION	Tasks run in parallel (e.g., download + UI update)
RESOURCE SHARING	Threads share memory space, making communication easier
ASYNCHRONOUS BEHAVIOR	Improves performance and user experience (e.g., in UI apps)

Single-threaded vs Multi-threaded

SINGLE-THREADED APP	MULTI-THREADED APP
ONE TASK AT A TIME	Multiple tasks at the same time
SLOWER AND LESS RESPONSIVE	Faster, more responsive

Thread vs Process

TERM	THREAD	PROCESS
DEFINITION	Smallest unit of a program	Independent program in memory
MEMORY	Shares memory with other threads	Has separate memory
OVERHEAD	Low	High

Real-life Examples of Multithreading:

- Web browsers: Render page + load resources + run JS
- Games: Background music + physics + rendering
- Text editor: Typing + spell check + autosave

Example Use Cases in Java:

- **Banking App**: One thread processes transactions, another logs them.
- **Video Player**: One thread decodes video, another handles audio.

5.1.2 Creating and Managing Threads in Java

Ways to Create a Thread in Java

There are **two main approaches**:

APPROACH	DESCRIPTION	USE WHEN...
1. EXTENDING THREAD	Create a subclass of Thread and override run()	You don't need to extend another class
2. IMPLEMENTING RUNNABLE	Create a class that implements Runnable and pass it to a Thread object	You need to extend another class

Method 1: Extending Thread Class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running using Thread class...");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // Create thread object
        t1.start();                   // Start thread
    }
}
Method 2: Implementing Runnable Interface
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running using Runnable interface...");
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t1 = new Thread(myRunnable); // Pass Runnable to Thread
        t1.start();
    }
}
```

Which One Should You Use?

- Prefer **Runnable** if your class already extends another class (since Java supports only single inheritance).
- Use **Thread** when you want to override Thread methods or don't need to extend any other class.

Creating Multiple Threads Example

```
class MyTask extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(getName() + ": " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyTask t1 = new MyTask();
        MyTask t2 = new MyTask();

        t1.start();
        t2.start();
    }
}
```

Practice Activities

1. Create a thread using Thread and Runnable—print your name 5 times in each.
2. Run 2 threads: One prints even numbers, the other prints odd numbers.
3. Modify the class to accept thread names and print them in the output.

5.1.3 Thread Lifecycle in Java

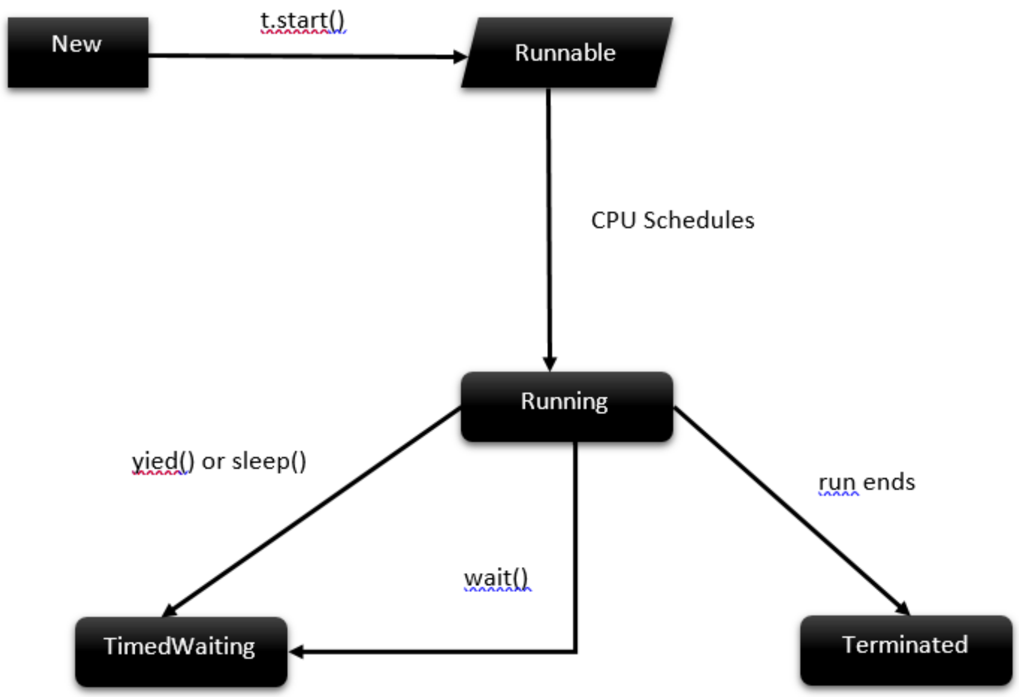
Analogy:

Think of a thread as a **task** performed by a **worker**.
Just like a worker goes through different stages—**hired**, **ready**, **working**, **waiting**, **done**—so does a thread.

Thread Lifecycle Stages

STATE	DESCRIPTION
NEW	Thread is created but not started
RUNNABLE	Thread is ready to run, waiting for CPU
RUNNING	Thread is executing
BLOCKED/WAITING	Thread is paused, waiting for a resource or another thread
TERMINATED	Thread has finished executing

Lifecycle Diagram



Description of Each State:

1. New

- Thread is created using new Thread() or by extending Thread.

Thread t = new Thread(); // NEW

2. Runnable

- You called start(). The thread is **ready**, waiting to be picked by CPU.

t.start(); // Moves to RUNNABLE

3. Running

- JVM scheduler has selected the thread to run.
- The thread's run() method is now executing.

4. Blocked/Waiting

- Thread is **waiting** due to:
  - sleep()
  - join()
  - wait()
- It resumes only when the condition is met (time ends, other thread completes, notify is called).

5. Terminated (Dead)

- run() method completes or an exception is thrown.

```
System.out.println("Thread done");
```

Lifecycle Demo in Code:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();  // NEW
        System.out.println(t.getState()); // NEW
        t.start();                    // RUNNABLE -> RUNNING
        System.out.println(t.getState()); // Might print RUNNABLE or TERMINATED
    }
}
```

5.1.4 Important Thread Methods in Java

These methods help you control how threads behave—when to pause, wait, yield control, or coordinate with others.

Commonly Used Thread Methods

METHOD	DESCRIPTION
START()	Starts a new thread (calls run() internally)
RUN()	Contains the task the thread will perform
SLEEP(MS)	Pauses the thread for a specific time
JOIN()	Waits for another thread to finish
YIELD()	Suggests that the current thread pause and let others run
ISALIVE()	Checks if a thread is still running
SETNAME() / GETNAME()	Sets or gets thread name

1. start() vs run()

```
Thread t = new Thread();
t.start(); // Executes in a new thread
t.run();   // Just a method call, no new thread
```

Always use start() to begin multithreaded execution.

2. sleep()

Pauses the current thread temporarily.

```
Thread.sleep(1000); // 1 second
```

InterruptedException must be handled using try-catch.

```
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    System.out.println("Interrupted!");
}
```

Use case: Delaying animations, retry mechanisms, simulating time.

3. join()

Waits for another thread to complete.

t1.join(); // Main thread waits for t1 to finish

Example:

```
Thread t1 = new Thread(() -> {
    for (int i = 0; i < 3; i++) {
        System.out.println("Child thread");
    }
});
t1.start();
t1.join(); // Main waits
System.out.println("Main thread runs after t1");
```

4. yield()

Temporarily pauses the current thread and allows other threads of the same priority to execute.

Thread.yield();

Not guaranteed to pause—it just **suggests** the CPU.

5. isAlive(), setName(), getName()

```
Thread t = new Thread();
t.setName("Worker-1");
System.out.println(t.getName());
System.out.println(t.isAlive()); // true if started and not finished
```

5.1.5 Synchronization in Java

Analogy:

Imagine two people trying to **withdraw money from the same ATM** at the same time. If they access the same account without taking turns, they might withdraw more than what's available — that's a **race condition**.

🔑 Solution? One person must wait — this is **synchronization**.

What is Synchronization?

Synchronization ensures that **only one thread can access a shared resource at a time**, preventing inconsistent or corrupt data.

The Problem Without Synchronization

```
class Counter {
    int count = 0;
    void increment() {
        count++;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for(int i = 0; i < 1000; i++) counter.increment();
        });
        Thread t2 = new Thread(() -> {
            for(int i = 0; i < 1000; i++) counter.increment();
        });

        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println("Final count: " + counter.count); // Expected: 2000
    }
}
```

Output may be less than 2000!
Why? Both threads try to update count at the same time.

Solution: Use synchronized

1. Synchronized Method

```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}
```

2. Synchronized Block

```
synchronized(counter) {
    counter.increment();
}
```

You can synchronize **only the critical section** (the part that modifies shared data), which is more efficient.

Use Cases

- Bank account operations
- Online booking systems
- Shared counters or lists

Locks Behind the Scenes

Every object in Java has a **monitor lock**. When a thread enters a synchronized method/block, it acquires the lock. Other threads trying to access it must wait.

5.1.6 Inter-thread Communication in Java

Analogy:

Imagine a **producer** (chef) preparing food and a **consumer** (waiter) serving it.

- If the chef is too fast, the waiter can't keep up.
- If the waiter is too fast, he may find nothing to serve.

They need to **coordinate**.

That's **inter-thread communication** — threads cooperating instead of competing.

Why It's Needed

Java threads can **pause and notify each other** using three main methods (defined in Object class):

METHOD	DESCRIPTION
WAIT()	Pauses the current thread
NOTIFY()	Wakes up a single waiting thread
NOTIFYALL()	Wakes up all waiting threads

Rules:

1. These methods must be called **within a synchronized block/method**
2. They must be called **on the same object** used for locking

Producer-Consumer Example (Simplified)

```
class Store {
    int item;
    boolean available = false;

    synchronized void produce(int value) {
        while (available) {
            try { wait(); } catch (InterruptedException e) {}
        }
        item = value;
        available = true;
        System.out.println("Produced: " + item);
        notify();  // Notify consumer
    }
```

```
    synchronized void consume() {
        while (!available) {
            try { wait(); } catch (InterruptedException e) {}
        }
        System.out.println("Consumed: " + item);
        available = false;
        notify();  // Notify producer
    }
}
public class Main {
    public static void main(String[] args) {
        Store store = new Store();

        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                store.produce(i);
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                store.consume();
            }
        });

        producer.start();
        consumer.start();
    }
}
```

Breakdown:

- **Producer waits** if item is already available.
- **Consumer waits** if there's nothing to consume.
- They **notify** each other after completing their action.

Key Notes:

- Use while (not if) to avoid **spurious wakeups**.
- wait() releases the lock; sleep() does not.
- Ideal for **resource sharing** situations.

5.2.1 Java Collections Framework Overview

What is the Java Collections Framework?

It's a **set of classes and interfaces** in Java that provides **ready-to-use data structures** (like lists, sets, maps) and algorithms (like sorting and searching).

Think of it as Java's built-in **toolbox** for managing groups of objects.

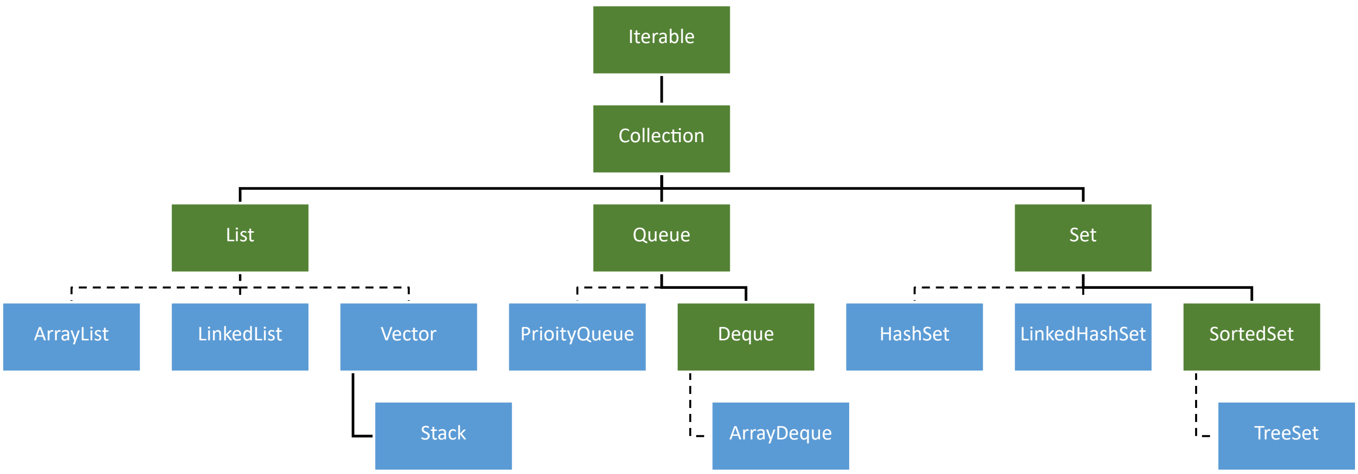
Why Use Collections?

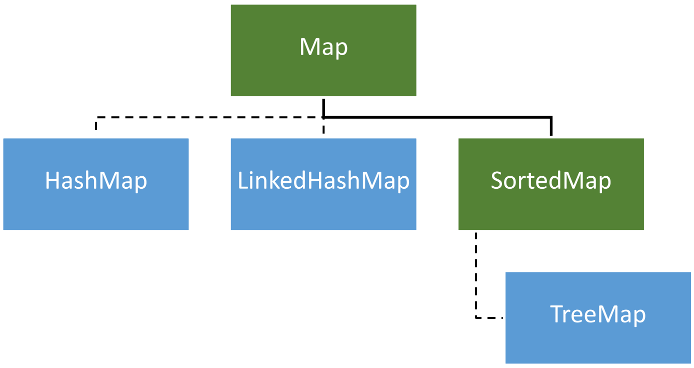
- No need to write your own data structures.
- Provides **efficient**, **scalable**, and **thread-safe** options.
- Interfaces allow **flexibility** and **interchangeability**.

Core Interfaces of Collections

INTERFACE	DESCRIPTION	COMMON IMPLEMENTATIONS
LIST	Ordered, duplicates allowed	ArrayList, LinkedList, Vector
SET	Unordered, no duplicates	HashSet, LinkedHashSet, TreeSet
MAP	Key-value pairs	HashMap, TreeMap, LinkedHashMap
QUEUE	FIFO structure	PriorityQueue, ArrayDeque

Collection Hierarchy





List vs Set vs Map

FEATURE	LIST	SET	MAP
ALLOWS DUPLICATES	Yes	❌ No	Keys no, Values yes
MAINTAINS ORDER	Yes (List)	Some (LinkedHashSet)	Yes (LinkedHashMap)
KEY ACCESS	❌	❌	Yes (via keys)

Key Classes at a Glance

1. ArrayList

- Resizable array
- Fast access, slow insertion/deletion in middle

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

2. LinkedList

- Nodes connected by links
- Good for frequent insertions/deletions

```
LinkedList<Integer> nums = new LinkedList<>();
nums.add(10);
nums.addFirst(5);
```

3. HashSet

- No duplicates, no ordering
- Uses hash table

```
HashSet<String> names = new HashSet<>();
names.add("John");
names.add("John"); // Ignored
```

4. HashMap

- Stores key-value pairs
- Keys must be unique

```
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "Apple");
map.put(2, "Mango");
System.out.println(map.get(1)); // Apple
Iterating over Collections
For List:
for (String item : list) {
    System.out.println(item);
}
```

For Map:

```
for (Map.Entry<Integer, String> entry : map.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

5.2.2 ArrayList in Java

1. Overview / Explanation

- ArrayList is a **resizable array** in Java (part of java.util).
- Maintains **insertion order**.
- Allows **duplicate elements**.
- Elements are indexed (like arrays).
- Ideal for **random access** and **frequent read** operations.

Use Case: Managing a dynamic list of items like names, scores, tasks.

2. Declaration

```
ArrayList<String> list;              // Generic declaration
List<Integer> numbers;               // Using interface type
```

3. Instantiation

```
list = new ArrayList<>();            // No initial size
numbers = new ArrayList<>(10);       // With initial capacity
```

Full example:

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

4. Adding Elements

```
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Mango");
fruits.add("Apple");  // Allows duplicates
```

5. Accessing Elements / Iteration

a) Index-based Access:

```
System.out.println(fruits.get(0)); // Apple
```

b) For Loop:

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println(fruits.get(i));
}
```

c) Enhanced For Loop:

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

d) forEach + Lambda (Java 8+):

```
fruits.forEach(f -> System.out.println(f));
```

6. Updating Elements

fruits.set(1, "Grapes"); // Replace Banana with Grapes

7. Deleting / Removing Elements

a) By Value:

```
fruits.remove("Apple"); // Removes first occurrence
```

b) By Index:

```
fruits.remove(0);        // Removes item at index 0
```

c) Remove All / Clear:

```
fruits.clear();          // Removes all elements
```

8. Searching / Contains Check

```
boolean hasMango = fruits.contains("Mango");
int index = fruits.indexOf("Apple");
```

9. Sorting

```
Collections.sort(fruits); // Sorts in ascending order
Descending:
fruits.sort(Collections.reverseOrder());
```

10. Other Useful Methods

```
int size = fruits.size();
boolean empty = fruits.isEmpty();
Object[] array = fruits.toArray();
```

5.2.3 LinkedList in Java

1. Overview / Explanation

- LinkedList is a **doubly-linked list** implementation of the List and Deque interfaces.
- Allows **duplicates** and **maintains insertion order**.
- Ideal for **frequent insertions and deletions** (especially in the middle or start).
- Slower than ArrayList for **random access** because there's no index-based storage internally.

Use Case: Implementing queues, playlists, undo functionality.

2. Declaration

```
LinkedList<String> list;             // Using class type
List<String> names;                  // Using interface type
```

3. Instantiation

```
list = new LinkedList<>();
names = new LinkedList<>();
```

Full example:

```
LinkedList<String> cities = new LinkedList<>();
```

4. Adding Elements

```
cities.add("Chennai");
cities.add("Mumbai");
cities.add("Delhi");

cities.addFirst("Kolkata");     // Adds at the beginning
cities.addLast("Bangalore");    // Adds at the end
```

5. Accessing Elements / Iteration

a) Index-based Access:

```
System.out.println(cities.get(2));
```

b) For Loop:

```
for (int i = 0; i < cities.size(); i++) {
    System.out.println(cities.get(i));
}
```

c) Enhanced For Loop:

```
for (String city : cities) {
    System.out.println(city);
}
```

d) Lambda with forEach:

```
cities.forEach(city -> System.out.println(city));
```

6. Updating Elements

```
cities.set(1, "Hyderabad");  // Replace Mumbai with Hyderabad
```

7. Deleting / Removing Elements

a) By Index:

```
cities.remove(3);
```

b) By Value:

```
cities.remove("Delhi");
```

c) Remove First/Last:

```
cities.removeFirst();
cities.removeLast();
```

d) Remove All:

```
cities.clear();
```

8. Searching / Contains Check

```
boolean found = cities.contains("Bangalore");
int index = cities.indexOf("Mumbai");
```

9. Sorting

```
Collections.sort(cities);
Descending:
cities.sort(Collections.reverseOrder());
```

10. Other Useful Methods

```
String first = cities.getFirst();
String last = cities.getLast();
int size = cities.size();
boolean empty = cities.isEmpty()
```

5.2.4 Stack in Java

1. Overview / Explanation

- Stack is a **Last-In-First-Out (LIFO)** data structure.
- Java provides Stack as a **class** in java.util (it extends Vector).
- You can also implement a stack using Deque (ArrayDeque is preferred in modern Java for stack operations due to performance).

Use Case: Undo operations, expression evaluation, backtracking, function calls.

2. Declaration

```
Stack<Integer> stack;
```

3. Instantiation

```
stack = new Stack<>();
Full example:
Stack<String> books = new Stack<>();
```

4. Adding Elements (Push)

```
books.push("Java");
books.push("Python");
books.push("C++");
```

5. Accessing Elements / Iteration

a) Iterating using loop:

```
for (String book : books) {
    System.out.println(book);
}
```

b) Access Top Element without Removing:

```
System.out.println(books.peek()); // Returns "C++"
```

6. Updating Elements

Stack doesn't offer direct updating methods (like set(index, value)), but if needed:

```
books.set(1, "C#"); // Replace at index 1
```

(Use with caution — this breaks typical stack usage)

7. Deleting / Removing Elements (Pop)

```
books.pop(); // Removes "C++"
You can also remove using:
books.remove("Python");
books.remove(0); // Removes by index
```

8. Searching / Contains Check

a) Contains:

```
books.contains("Java");
```

b) Search position from top (1-based index):

```
int pos = books.search("Java"); // 2
```

9. Sorting

Stacks are not meant to be sorted, but it can be done using:

```
Collections.sort(books);
```

(Note: this violates LIFO nature — use only if needed for special cases.)

10. Other Useful Methods

```
boolean empty = books.isEmpty();
int size = books.size();
books.clear();
```

5.2.5 PriorityQueue in Java

1. Overview / Explanation

- PriorityQueue is a **queue** that retrieves elements based on their **priority** rather than the order they were added.
- By default, it behaves like a **Min-Heap** (smallest element has the highest priority).
- It does **not allow null** elements.
- Not thread-safe (use PriorityBlockingQueue for concurrency).

Use Case: Task schedulers, Dijkstra's algorithm, bandwidth management, etc.

2. Declaration

```
PriorityQueue<Integer> pq;
Queue<String> taskQueue;
```

3. Instantiation

```
pq = new PriorityQueue<>();
taskQueue = new PriorityQueue<>();
Custom comparator (e.g., for Max-Heap):
PriorityQueue<Integer> maxPQ = new PriorityQueue<>(Collections.reverseOrder());
```

4. Adding Elements

```
pq.add(10);
pq.add(5);
pq.add(15);
pq.add(1);
```

Note: Elements are reordered internally to maintain heap property, not insertion order.

5. Accessing Elements / Iteration

a) Peek (retrieve head without removal):

```
System.out.println(pq.peek()); // Will show the smallest element
```

b) Iteration (order not guaranteed):

```
for (int num : pq) {
    System.out.println(num);
}
```

6. Updating Elements

There is **no direct update** method. Remove the element and re-add the updated value.

```
pq.remove(10);
pq.add(12);
```

7. Deleting / Removing Elements

a) Remove head:

```
pq.poll(); // Removes smallest element
```

b) Remove specific element:

```
pq.remove(15);
```

c) Clear all:

```
pq.clear();
```

8. Searching / Contains Check

```
boolean hasFive = pq.contains(5);
```

9. Sorting

Not applicable directly as PriorityQueue manages its internal order based on priority.

To sort, extract elements into a list:

```
List<Integer> sortedList = new ArrayList<>();
while (!pq.isEmpty()) {
    sortedList.add(pq.poll());
}
```

10. Other Useful Methods

```
int size = pq.size();
boolean empty = pq.isEmpty();
Object[] arr = pq.toArray();
```

5.2.6 ArrayDeque in Java

1. Overview / Explanation

- ArrayDeque (Array Double-Ended Queue) is a **resizable array-based implementation** of the Deque interface.
- Allows insertion and deletion from **both ends** (head and tail).
- **Faster than Stack and LinkedList** for stack/queue operations.
- Does **not allow null** elements.
- Can function as:
    - **Queue** (FIFO)
    - **Stack** (LIFO)

Use Case: Undo-redo stack, browser forward/back navigation, queue of tasks.

2. Declaration

```
Deque<String> deque;
ArrayDeque<Integer> intDeque;
```

3. Instantiation

```
deque = new ArrayDeque<>();
intDeque = new ArrayDeque<>(10); // Optional initial capacity
```

4. Adding Elements

a) As Queue:

```
deque.addLast("A");
deque.addLast("B");
deque.addLast("C");
```

b) As Stack:

```
deque.addFirst("X");
deque.addFirst("Y");
```

Other methods:

```
deque.offer("Z");           // Add to tail
deque.offerFirst("Start"); // Add to head
deque.offerLast("End");     // Add to tail
```

5. Accessing Elements / Iteration

a) Peek First and Last:

```
System.out.println(deque.peekFirst());
System.out.println(deque.peekLast());
```

b) Iterate:

```
for (String item : deque) {
    System.out.println(item);
}
```

6. Updating Elements

Like PriorityQueue, **no direct update**; remove and re-insert.

```
deque.remove("A");
deque.add("A_updated");
```

7. Deleting / Removing Elements

```
deque.removeFirst(); // Removes head
deque.removeLast();  // Removes tail
deque.poll();        // Removes head, returns null if empty
deque.clear();       // Clears entire deque
```

8. Searching / Contains Check

```
boolean exists = deque.contains("B");
```

9. Sorting

You can convert to a list and sort:

```
List<String> list = new ArrayList<>(deque);
Collections.sort(list);
```

Then rebuild the deque if needed:

```
deque = new ArrayDeque<>(list);
```

10. Other Useful Methods

```
int size = deque.size();
boolean empty = deque.isEmpty();
```

5.2.7 HashSet in Java

1. Overview / Explanation

- HashSet is a part of Java's Collection Framework that implements the **Set** interface.
- It stores **unique elements only** — no duplicates allowed.
- **No guaranteed order** (not insertion order or sorted).
- Backed by a **hash table**.
- Allows **null** (only one null element).

Use Case: Removing duplicates, membership testing, set operations like union/intersection.

2. Declaration

```
Set<String> names;
HashSet<Integer> numbers;
```

3. Instantiation

```
names = new HashSet<>();
numbers = new HashSet<>(20);  // with initial capacity
```

Full example:

```
HashSet<String> fruits = new HashSet<>();
```

4. Adding Elements

```
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Orange");
fruits.add("Apple"); // Duplicate - will be ignored
```

5. Accessing Elements / Iteration

a) Enhanced for-loop:

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

b) Iterator:

```
Iterator<String> itr = fruits.iterator();
while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

6. Updating Elements

There is **no direct update** in a set. You need to remove the old value and add a new one:

```
fruits.remove("Orange");
fruits.add("Mango");
```

7. Deleting / Removing Elements

```
fruits.remove("Banana");
fruits.clear(); // removes all elements
```

8. Searching / Contains Check

```
boolean hasApple = fruits.contains("Apple");
```

9. Sorting

Since HashSet is **unordered**, you must convert it to a list to sort:

```
List<String> sortedFruits = new ArrayList<>(fruits);
Collections.sort(sortedFruits);
System.out.println(sortedFruits);
```

10. Other Useful Methods

```
int size = fruits.size();
boolean empty = fruits.isEmpty();
```

5.2.8 LinkedHashSet in Java

1. Overview / Explanation

- LinkedHashSet is a **HashSet** with a **predictable iteration order**.
- It **maintains insertion order** using a **doubly-linked list** internally.
- Like HashSet, it:
    - Stores **unique elements only** (no duplicates)
    - Allows **one null**
    - Is **not synchronized**

Use Case: When you want a set with **no duplicates** but also need to **preserve the insertion order**.

2. Declaration

```
Set<String> set;
LinkedHashSet<Integer> numbers;
```

3. Instantiation

```
set = new LinkedHashSet<>();
numbers = new LinkedHashSet<>(20); // with initial capacity
```

Example:

```
LinkedHashSet<String> colors = new LinkedHashSet<>();
```

4. Adding Elements

```
colors.add("Red");
colors.add("Green");
colors.add("Blue");
colors.add("Red"); // Duplicate, will be ignored
```

5. Accessing Elements / Iteration

Maintains the **order in which elements were added**.

```
for (String color : colors) {
    System.out.println(color);
}
```

Or using iterator:

```
Iterator<String> it = colors.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

6. Updating Elements

Like HashSet, there's **no direct update**. You must remove and re-add the element.

```
colors.remove("Blue");
colors.add("Cyan");
```

7. Deleting / Removing Elements

```
colors.remove("Green");
colors.clear(); // removes all elements
```

8. Searching / Contains Check

```
boolean hasRed = colors.contains("Red");
```

9. Sorting

To sort, convert it to a list:

```
List<String> sortedColors = new ArrayList<>(colors);
Collections.sort(sortedColors);
System.out.println(sortedColors);
```

10. Other Useful Methods

```
int size = colors.size();
boolean empty = colors.isEmpty();
```

5.2.9 TreeSet in Java

1. Overview / Explanation

- TreeSet is a **SortedSet** implementation that stores elements in **ascending order** by default.
- Uses a **Red-Black Tree** internally.
- **No duplicates allowed**
- Does **not allow null** elements (unlike HashSet).
- Provides methods to access elements based on sorting order.

Use Case: When you need a **unique set of elements that are automatically sorted**.

2. Declaration

```
Set<Integer> set;
TreeSet<String> cities;
```

3. Instantiation

```
set = new TreeSet<>();
cities = new TreeSet<>();
```

For custom sorting (e.g., descending order):

```
TreeSet<Integer> descSet = new TreeSet<>(Collections.reverseOrder());
```

4. Adding Elements

```
cities.add("Delhi");
cities.add("Mumbai");
cities.add("Chennai");
cities.add("Delhi"); // Duplicate - will be ignored
```

5. Accessing Elements / Iteration

Elements will be returned in **sorted order**:

```
for (String city : cities) {
    System.out.println(city);
}
```

Or using iterator:

```
Iterator<String> itr = cities.iterator();
while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

6. Updating Elements

No direct update; remove the old one and add the updated:

```
cities.remove("Chennai");
cities.add("Hyderabad");
```

7. Deleting / Removing Elements

```
cities.remove("Mumbai");
cities.clear(); // remove all elements
```

8. Searching / Contains Check

```
boolean found = cities.contains("Delhi");
```

9. Sorting

Not needed – TreeSet is **always sorted**.

```
Custom sorting (e.g., reverse alphabetical):
TreeSet<String> reverseCities = new TreeSet<>(Collections.reverseOrder());
reverseCities.addAll(cities);
```

10. Other Useful Methods

```
int size = cities.size();
boolean empty = cities.isEmpty();
```

Additional Navigational Methods:

```
System.out.println(cities.first());   // Smallest
System.out.println(cities.last());    // Largest
System.out.println(cities.higher("Delhi")); // Next greater element
System.out.println(cities.lower("Delhi"));  // Previous smaller element
```

5.2.10 Conversions Involving Java Collections

1. Array ➡️ List

```
String[] fruits = {"Apple", "Banana", "Mango"};
List<String> fruitList = Arrays.asList(fruits);
Note: Arrays.asList() returns a fixed-size list backed by the array. To make it
resizable:
List<String> resizableList = new ArrayList<>(Arrays.asList(fruits));
```

2. List ➡️ Array

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
String[] array = list.toArray(new String[0]);
```

3. List ➡️ Set

```
List<String> names = Arrays.asList("Ravi", "Ravi", "Kiran");
Set<String> uniqueNames = new HashSet<>(names); // removes duplicates
```

4. Set ➡️ List

```
Set<String> colors = new HashSet<>();
colors.add("Red");
colors.add("Blue");

List<String> colorList = new ArrayList<>(colors);
```

5. Set ➡️ Array

```
Set<Integer> numbers = new HashSet<>();
numbers.add(1);
numbers.add(2);

Integer[] numberArray = numbers.toArray(new Integer[0]);
```

6. Array ➡️ Set

```
String[] names = {"Ravi", "Ravi", "Anil"};
Set<String> nameSet = new HashSet<>(Arrays.asList(names));
```

7. Map ➡️ Set (of Keys / Values / Entries)

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "A");
map.put(2, "B");

Set<Integer> keys = map.keySet();
Collection<String> values = map.values();
Set<Map.Entry<Integer, String>> entries = map.entrySet();
```

8. Set (of entries) ➡️ Map

```
Set<Map.Entry<Integer, String>> entries = map.entrySet();
Map<Integer, String> newMap = new HashMap<>();

for (Map.Entry<Integer, String> entry : entries) {
    newMap.put(entry.getKey(), entry.getValue());
}
```

9. List ➡️ Map (with unique keys)

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

Map<Integer, String> studentMap = new HashMap<>();
for (int i = 0; i < students.size(); i++) {
    studentMap.put(i + 1, students.get(i)); // RollNo => Name
}
```

5.2.11 HashMap in Java

1. Overview / Explanation

- HashMap is a part of the Java Collection Framework.
- Stores **key-value pairs**.
- **No duplicate keys** (keys must be unique, but values can repeat).
- Allows **one null key** and multiple null values.
- **Unordered** – does not maintain insertion or sorted order.
- Internally uses a **hash table** for fast access.

Use Case: Storing mappings like studentID → studentName, productCode → price, etc.

2. Declaration

```
Map<Integer, String> studentMap;
HashMap<String, Integer> ageMap;
```

3. Instantiation

```
studentMap = new HashMap<>();
ageMap = new HashMap<>();
```

4. Adding Elements (put)

```
studentMap.put(101, "Ravi");
studentMap.put(102, "Anu");
studentMap.put(103, "Kiran");
studentMap.put(101, "Raj");  // Overwrites value for key 101
```

5. Accessing Elements (get & iteration)

```
System.out.println(studentMap.get(102));  // Output: Anu
Iteration over entries:
for (Map.Entry<Integer, String> entry : studentMap.entrySet()) {
    System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
}
Iteration using keySet:
for (Integer key : studentMap.keySet()) {
    System.out.println("Key: " + key + ", Value: " + studentMap.get(key));
}
```

6. Updating Elements

```
studentMap.put(103, "Karthik"); // replaces "Kiran"
```

7. Deleting Elements

```
studentMap.remove(102);
studentMap.clear(); // removes all entries
```

8. Searching / Contains Check

```
studentMap.containsKey(101);   // true
studentMap.containsValue("Anu"); // true or false
```

9. Sorting

Since HashMap is **unordered**, to sort:

a) By keys (ascending):

```
Map<Integer, String> sortedByKey = new TreeMap<>(studentMap);
```

b) By values:

```
studentMap.entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue())
    .forEach(System.out::println);
```

10. Other Useful Methods

```
studentMap.size();
studentMap.isEmpty();
```

5.2.12 LinkedHashMap in Java

1. Overview / Explanation

- LinkedHashMap is a **Map** implementation that **preserves the insertion order**.
- Inherits from HashMap, but uses a **doubly-linked list** to maintain order.
- Allows **one null key** and multiple null values.
- **Faster iteration** compared to HashMap because of predictable order.

Use Case: When you need a key-value mapping **with predictable insertion order**.

2. Declaration

```
Map<Integer, String> studentMap;
LinkedHashMap<String, Integer> ageMap;
```

3. Instantiation

```
studentMap = new LinkedHashMap<>();
ageMap = new LinkedHashMap<>();
```

Optional: Create with **access-order** (for LRU cache-like behavior):

```
LinkedHashMap<Integer, String> lruMap = new LinkedHashMap<>(16, 0.75f, true);
```

4. Adding Elements (put)

```
studentMap.put(101, "Ravi");
studentMap.put(102, "Anu");
studentMap.put(103, "Kiran");
studentMap.put(101, "Raj");  // Overwrites value for key 101
```

📝 **Insertion order is maintained**:

```
101=Raj, 102=Anu, 103=Kiran
```

5. Accessing Elements

```
System.out.println(studentMap.get(102));  // Output: Anu
Iteration (in insertion order):
for (Map.Entry<Integer, String> entry : studentMap.entrySet()) {
    System.out.println(entry.getKey() + " => " + entry.getValue());
}
```

6. Updating Elements

```
studentMap.put(103, "Karthik"); // updates "Kiran"
```

7. Deleting Elements

```
studentMap.remove(101);
studentMap.clear(); // removes all entries
```

8. Searching / Contains Check

```
studentMap.containsKey(102);      // true
studentMap.containsValue("Ravi"); // false
```

9. Sorting

If you need to sort:

By keys:

```
Map<Integer, String> sorted = new TreeMap<>(studentMap);
```

By values (using stream):

```
studentMap.entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue())
    .forEach(System.out::println);
```

10. Other Useful Methods

```
studentMap.size();
studentMap.isEmpty();
studentMap.keySet();
studentMap.values();
```

5.2.13 TreeMap in Java

1. Overview / Explanation

- TreeMap is a **Map** implementation that keeps **keys sorted** in **natural order** (or by a custom comparator).
- Uses a **Red-Black Tree** internally.
- **No duplicate keys allowed**.
- **Does not allow null keys** (unlike HashMap), but allows multiple null values.
- Slower than HashMap, but useful when **sorted keys** are required.

Use Case: Whenever you need a **sorted key-value mapping** (e.g., student marks by roll number, product catalog sorted by code).

2. Declaration

```
Map<Integer, String> treeMap;
TreeMap<String, Integer> marksMap;
```

3. Instantiation

```
treeMap = new TreeMap<>();
marksMap = new TreeMap<>();
```

For **custom sorting** (e.g., reverse order):

```
TreeMap<Integer, String> reverseMap = new TreeMap<>(Collections.reverseOrder());
```

4. Adding Elements (put)

```
treeMap.put(103, "Ravi");
treeMap.put(101, "Anu");
treeMap.put(102, "Kiran");
treeMap.put(104, "Raj");
```

🔁 **Automatically sorted by keys**:

```
101=Anu, 102=Kiran, 103=Ravi, 104=Raj
```

5. Accessing Elements

```
System.out.println(treeMap.get(102));  // Output: Kiran
```

Iteration (Sorted Order):

```
for (Map.Entry<Integer, String> entry : treeMap.entrySet()) {
    System.out.println(entry.getKey() + " => " + entry.getValue());
}
```

6. Updating Elements

```
treeMap.put(102, "Karthik"); // updates "Kiran"
```

7. Deleting Elements

```
treeMap.remove(103);
treeMap.clear(); // remove all entries
```

8. Searching / Contains Check

```
treeMap.containsKey(101);       // true
treeMap.containsValue("Anu");   // true
```

9. Sorting

Already **sorted by keys**. For **custom sorting**, you can use:

```
TreeMap<String, Integer> customMap = new TreeMap<>(Comparator.reverseOrder());
customMap.putAll(marksMap);
```

To sort **by values**, use streams:

```
treeMap.entrySet()
       .stream()
       .sorted(Map.Entry.comparingByValue())
       .forEach(System.out::println);
```

10. Other Useful Methods

```
treeMap.firstKey();      // smallest key
treeMap.lastKey();       // largest key
treeMap.higherKey(102); // next greater key
treeMap.lowerKey(102);  // previous smaller key

treeMap.keySet();
treeMap.values();
```

5.2.14 Comparable vs Comparator in Java

Both are used to compare and sort objects, but they differ in how and where the sorting logic is defined.

1. Comparable Interface (Natural Ordering)

Key Points:

- Found in java.lang
- Must override compareTo()
- Sorting logic is part of the class itself
- Used when a class has a **natural/default ordering**

Syntax:

```
public class Student implements Comparable<Student> {
    int id;
    String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public int compareTo(Student s) {
        return this.id - s.id;  // Ascending order by ID
    }
}
```

Example:

```
List<Student> list = new ArrayList<>();
list.add(new Student(102, "Ravi"));
list.add(new Student(101, "Amit"));

Collections.sort(list);  // uses compareTo()
```

2. Comparator Interface (Custom Ordering)

Key Points:

- Found in java.util
- Used for **external or multiple sort strategies**
- Override compare()
- Often used with lambda expressions

Syntax:

```
class NameComparator implements Comparator<Student> {
    public int compare(Student a, Student b) {
        return a.name.compareTo(b.name);
    }
}
```

Example:

Collections.sort(list, new NameComparator());

Java 8+ Lambda version:

Collections.sort(list, (a, b) -> a.name.compareTo(b.name));

Difference Table

FEATURE	COMPARABLE	COMPARATOR
PACKAGE	java.lang	java.util
METHOD	compareTo(T o)	compare(T o1, T o2)
DEFINES IN	Same class	Separate class or lambda
SORTING TYPE	Natural (default)	Custom (flexible)
AFFECTS	One default sorting logic	Multiple sorting criteria possible
USAGE	Collections.sort(list)	Collections.sort(list, comparator)

Real-Life Analogy

Think of a Student class:

- **Comparable**: "Sort by student roll number" — default logic inside the class.
- **Comparator**: "Sort by name, then by marks, then by DOB" — various strategies based on situation.

Example: Sort by ID, then by Name

```
Collections.sort(list, Comparator
    .comparing(Student::getId)
    .thenComparing(Student::getName));
```