# Java Backend Development Live-85

lecture-5

# Agenda

- Built-in functional interfaces: Predicate, Consumer, Function
- Java Streams
- DSA Problems (based on Collections)
- Multithreading in Java (Introduction)
- Thread and Runnable
- Executors Framework
- ExecutorService & ThreadPoolExecutor

# Predicate<T>

- **Purpose**: Represents a boolean condition on a single argument.
- **Method**: `boolean test(T t)`
- **Usage**: Commonly used for filtering and matching elements.
- **Useful Methods:**
  - `and(Predicate)`: Combines two predicates with logical AND.
  - `or(Predicate)`: Combines two predicates with logical OR.
  - `negate()`: Negates the current predicate.

```
Predicate<Integer> isEven = n -> n % 2 == 0;
System.out.println(isEven.test( t: 4)); // Output: true
System.out.println(isEven.test( t: 5)); // Output: false
```

# Consumer<T>

- **Purpose**: Represents an operation that accepts a single input and returns no result.
- **Method**: `void accept(T t)`
- **Usage**: Performing actions like logging, printing, or modifying input.
- **Useful Methods:**
  - `andThen(Consumer)`: Chains multiple consumers to execute in sequence.

```
Consumer<String> printMessage = message -> System.out.println("Message: " + message);
printMessage.accept( t: "Hello, World!"); // Output: Message: Hello, World!
```

# Function<T,R>

- **Purpose**: Represents a function that takes one argument and produces a result.
- **Method**: `R apply(T t)`
- **Usage**: Transforming or mapping data.
- **Useful Methods:**
  - `andThen(Function)`: Chains another function to execute after the current one.
  - `compose(Function)`: Executes another function before the current one.

```
Function<Integer, String> intToString = n -> "Number: " + n;
System.out.println(intToString.apply( t: 10)); // Output: Number: 10
```

# Supplier<T>

- **Purpose**: Represents a supplier of results (no input, only output).
- **Method**: `T get()`
- **Usage**: Used for lazy initialization or providing default values.

```
Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get()); // Output: Random value (e.g., 0.8473)
```

# Comparison

| Interface | Input Type(s) | Output Type | Purpose | Key Method |
|---|---|---|---|---|
| `Predicate<T>` | `T` | `boolean` | Tests a condition. | `test(T t)` |
| `Consumer<T>` | `T` | `void` | Performs an action on input. | `accept(T t)` |
| `Function<T,R>` | `T` | `R` | Transforms input to output. | `apply(T t)` |
| `Supplier<T>` | `None` | `T` | Supplies a result. | `get()` |

# Java Streams API

- **Java Stream** is a sequence of elements supporting **functional-style operations**.
- **Stream vs Collections**
- Stream does **not store data**; it operates on the source (e.g., a collection).
- Stream operations are **lazy** and evaluated only when needed.
- **Composability**: Easily chain multiple operations.
- **Parallelism**: Support for parallel execution.
- Simplifies complex data manipulation.
- Avoids boilerplate loops.

# Stream API Workflow

- **Source**: Start with a data source (e.g., List, Set, Array).
- **Intermediate Operations**: Transform the data (e.g., `filter`, `map`).
- **Terminal Operations**: Produce a result (e.g., `collect`, `forEach`).

```java
public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    // Filter even numbers and double them
    List<Integer> result = numbers.stream()
            .filter(n -> n % 2 == 0)
            .map(n -> n * 2)
            .collect(Collectors.toList());
    System.out.println(result); // Output: [4, 8]
}
```

# DSA Problems (based on Collections)

- Problem1:https://practice.geeksforgeeks.org/problems/first-repeating-element4018/1/
- Problem 2: https://practice.geeksforgeeks.org/problems/valid-expression1025/1/
- Problem 3: https://practice.geeksforgeeks.org/problems/k-largest-elements4206/1/
- Problem 4: https://practice.geeksforgeeks.org/problems/level-order-traversal/1/
- Problem 5: https://practice.geeksforgeeks.org/problems/key-pair5616/1/

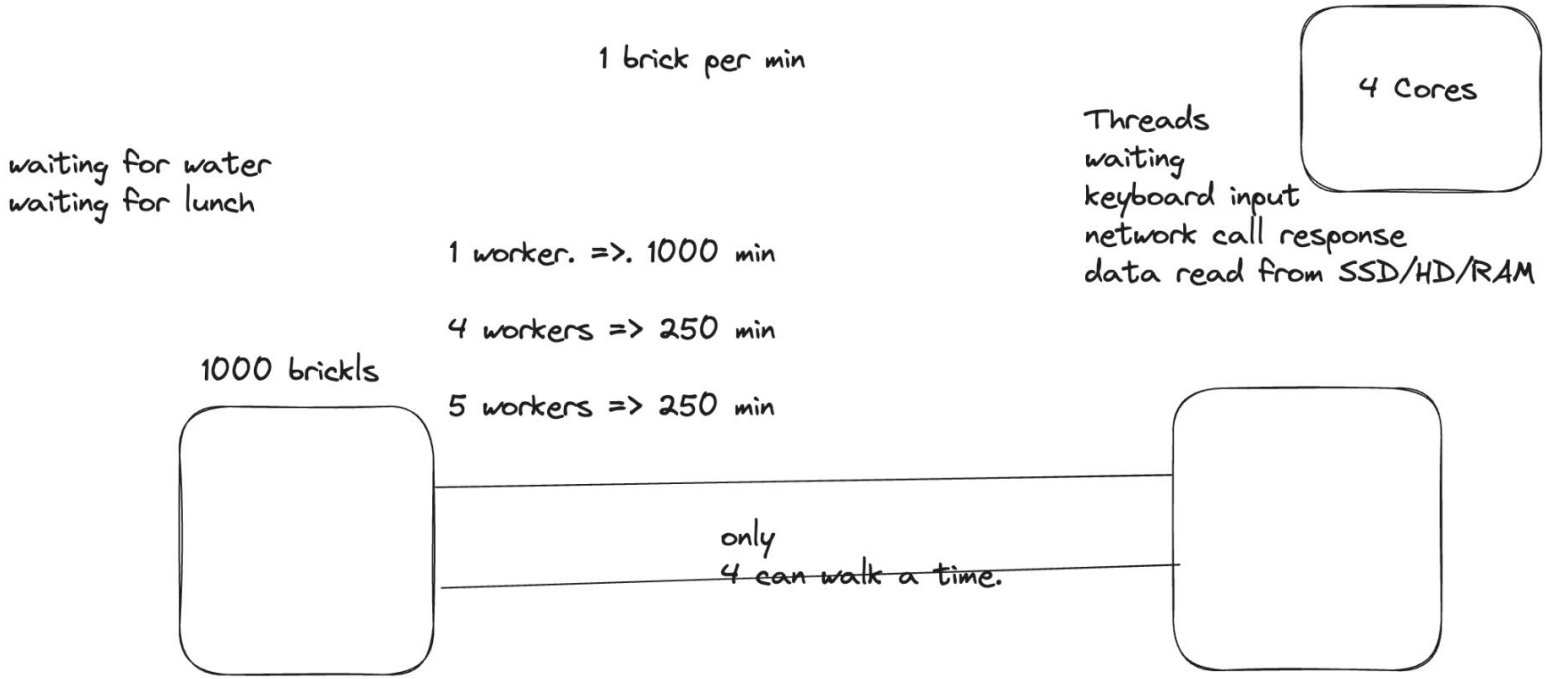
# Multithreading in Java

**Thread:**

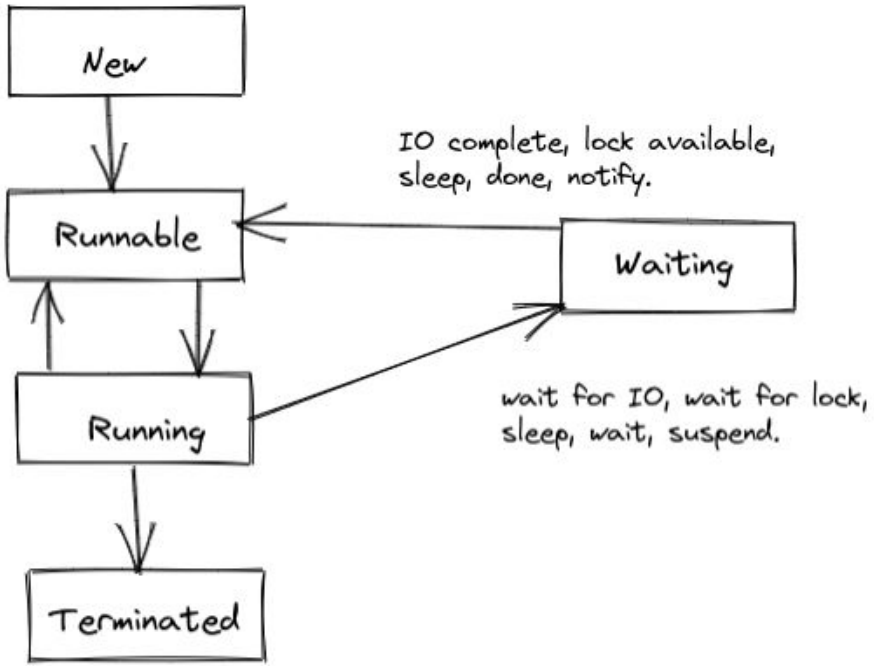
A thread is actually a lightweight process. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called thread and each thread defines a separate path of execution.

A Thread is created & controlled by the java.lang.Thread class.

# Real Life Example



# Java Thread Life Cycle




Life Cycle of Java Thread

# Main Thread in Java

It is created automatically when your program is started

Main method represents the execution path of main thread.

Long running task can block main thread, introduce slowness in App, or hang the App.

We use child thread(worker thread) to do other work.

Every thread have its own Stack.

# Create Thread in Java (Thread Class)

We can create a thread by creating a new class that extends Thread class, then override the run() method and then create an instance of that class.

```
public class MyThread extends Thread {

    public void run(){

    System.out.println("MyThread running");

  }

}

MyThread t1 = new MyThread ();

t1.start();
```

# Create Thread in Java (Runnable Interface)

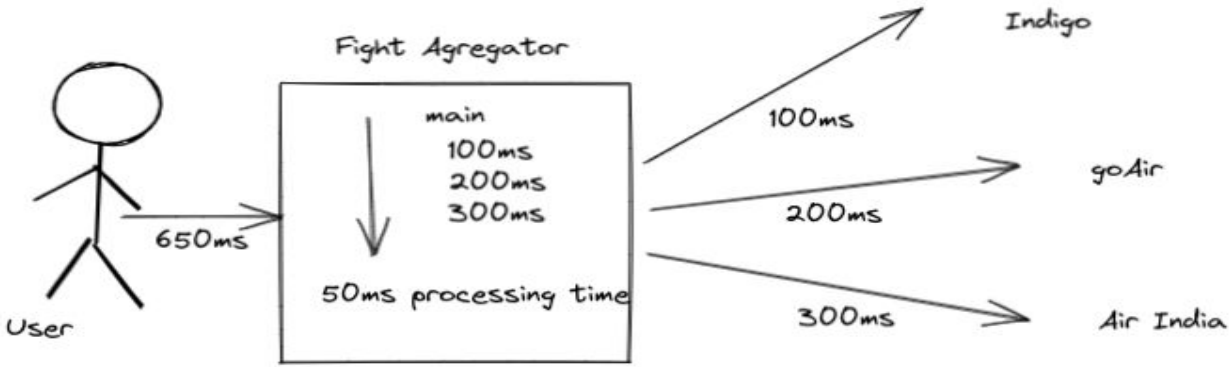
Another way to create a thread is to create a class that implements the **Runnable** interface.

```
public class MyTask implements Runnable {

public void run(){

System.out.println("MyTask running");

   }

}

Thread t1 = new Thread(new MyTask ());

t1.start();
```

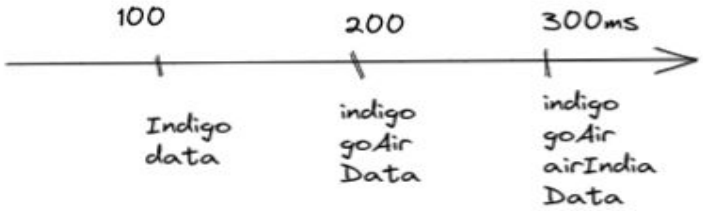
# Thread vs Runnable

| **Thread Class** | **Runnable Interface** |
| --- | --- |
| Thread is a class. It is used to create a thread | Runnable is a functional interface which is used to create a thread (task/work) |
| It has multiple methods including start() and run() | It has only abstract method run() |
| Each thread creates a unique object and gets associated with it | Multiple threads share the same objects. |
| More memory required | Less memory required |
| Multiple Inheritance is not allowed in java hence after a class extends Thread class, it can not extend any other class | If a class is implementing the runnable interface then that class can extend another class as well. |

# Example: flight aggregator




Using 3 thread.
300ms All data will available.
API response time:350ms



# Executors Framework (Thread Pool)

A framework for creating and managing threads.

- **Thread Creation**: It provides various methods for creating threads and pool of threads.
- **Thread Management**: It manages the life cycle of the threads in the thread pool.
- **Task submission and execution**: Executors framework provides methods for submitting tasks for execution in the thread pool.

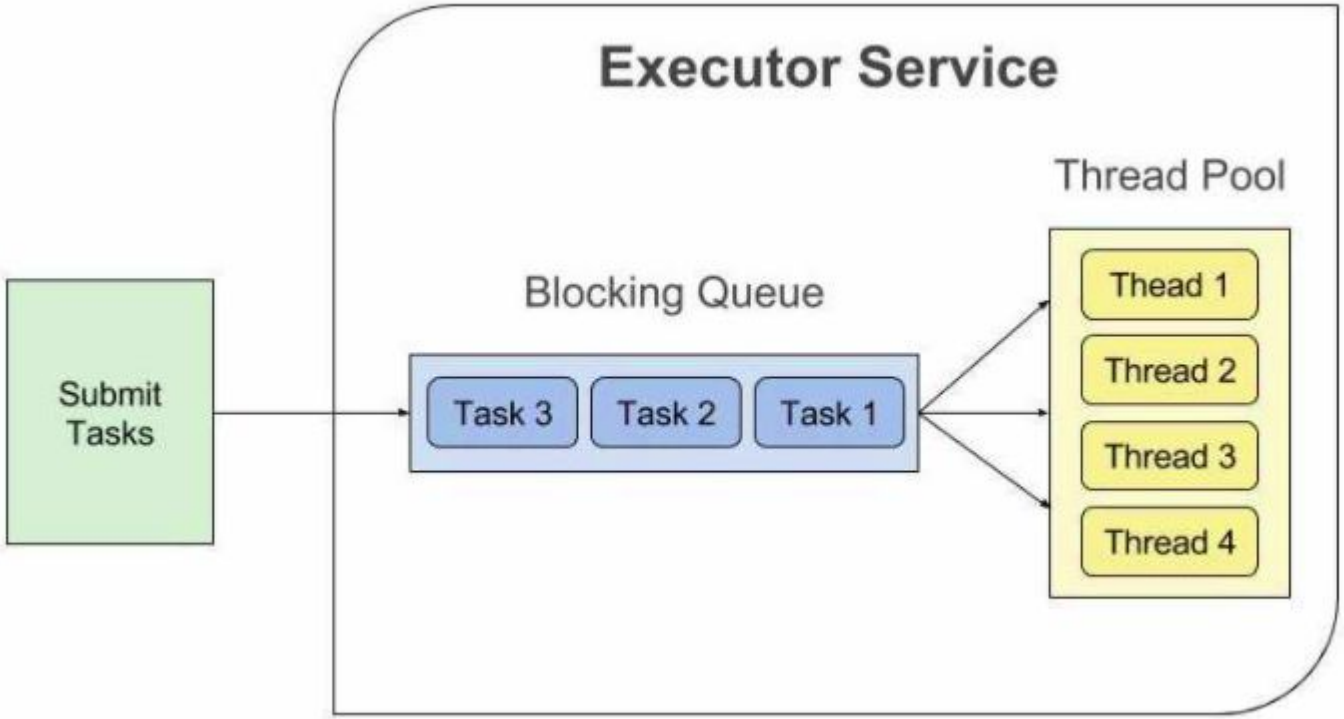
# ExecutorService

```
ExecutorService fixExecutorService = Executors.newFixedThreadPool(5);
ExecutorService executorService = Executors.newSingleThreadExecutor();
        executorService.submit(() -> {
            System.out.println("Task Running in : " + Thread.currentThread().getName());
        });
        executorService.shutdown();
```

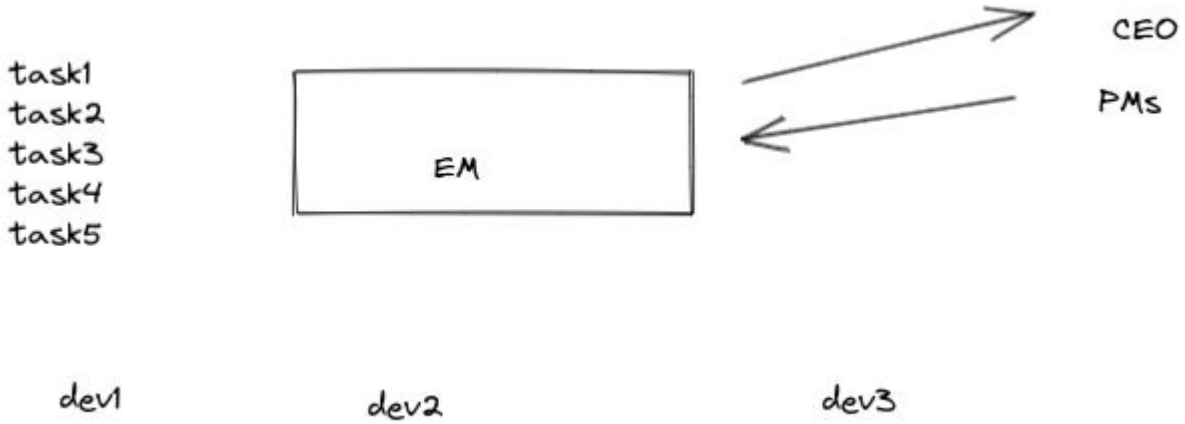
- shutdown() - when `shutdown()` method is called on an executor service, it stops accepting new tasks, waits for previously submitted tasks to execute, and then terminates the executor.
- shutdownNow() - this method interrupts the running task and shuts down the executor immediately.

# Thread Pool

- Creating a thread is an expensive operation and it should be minimized.
- Having worker threads minimizes the overhead due to thread creation because executor service has to create the thread pool only once and then it can reuse the threads for executing any task.
- Tasks are submitted to a thread pool via an internal queue called the ***Blocking Queue***.


Executor Service
Thread Pool
Blocking Queue
Submit Tasks
Task 3
Task 2
Task 1
Thead 1
Thread 2
Thread 3
Thread 4

Engineering Manager as ExecutorService and developers as worker threads.



# ThreadPoolExecutor

```
int  corePoolSize  =   5;
int  maxPoolSize   =   10;
long keepAliveTime = 5000;
ExecutorService threadPoolExecutor =
        new ThreadPoolExecutor(
                corePoolSize,
                maxPoolSize,
                keepAliveTime,
                TimeUnit.MILLISECONDS,
                new LinkedBlockingQueue<Runnable>()
        );
```