## **Thread Pools and Executors**

In the last lesson, we learned how to create threads in Java using two ways: by extending the **Thread** class or by using the **Runnable** interface.

Making one or two threads is easy, but when an application needs 20 or 30 threads to run at the same time, it becomes harder to manage.

In big applications, there can be hundreds or even thousands of threads running together. So, it’s helpful to keep thread creation and management separate from the main part of the program.

**Let’s meet the Executors Framework**

The **Executors Framework** in Java helps us to create and manage threads easily.

**And here's how it works:**

* **Thread Creation**: It provides tools to create a group of threads, called a **thread pool**, so our program can run tasks at the same time.
* **Thread Management**: It takes care of the threads in the pool, so we don't need to worry about whether they are busy, active, or done before running a task.
* **Task Submission and Execution**: It helps us to submit tasks for execution in the thread pool. We can choose when a task will run, whether immediately, later, or repeatedly.

In **Java Concurrency API** we have three key interfaces for working with threads:

* **Executor**: This basic interface has a method called **execute()**, which runs a task that we define using Runnable.
* **ExecutorService**: This is an improved version of Executor, with extra features to manage the tasks' life cycle. It has a **submit()** method that can accept both **Runnable** and **Callable**. Callable tasks are similar to Runnable tasks but can return a result.

The **ExecutorService** framework in Java manages a pool of worker threads and assigns tasks to them. It abstracts away the complexity of creating and managing threads directly.

**Real World Analogy:** Consider a task manager who delegates work to available team members instead of hiring someone new for every task.

* **ScheduledExecutorService**: This interface is a more advanced version of ExecutorService that lets us schedule tasks to run at specific times or intervals.

**What is a Thread Pool ?**

A thread pool is a group of worker threads that are separate from the tasks they need to do, like Runnable tasks. This group of threads is managed by an executor.

## **Real-World Analogy:** Think of a thread pool like a team of chefs in a restaurant. The chefs (worker threads) are always ready to cook, but they don’t decide what to make. Instead, they wait for orders (tasks) from the customers. When an order comes in, one of the chefs picks it up and starts cooking. This way, the restaurant can serve many customers quickly without needing to hire new chefs every time an order comes in.

**Why Use Thread Pools ?**Imagine you own a small cafe. Each time a customer orders, you need a worker to serve them. Hiring a new worker for every single customer is inefficient, expensive, and slow. Instead, you hire a few workers who can handle multiple customers throughout the day. This idea is similar to using thread pools in programming. In Java, thread pools reuse threads, making it efficient when we have multiple tasks to handle.

**Usage Example of Thread Pools:** In web servers, each incoming request is a task that needs a thread. Using a Thread Pool means the server can handle many requests at once without slowing down.

**Combination of ExecutorService and Thread Pools** ExecutorService is like a manager in the cafe, assigning tasks to available workers. Instead of creating new threads (or “workers”) for each task, it manages a pool of threads, reusing them as tasks come in.

**ExecutorService** Is a Java framework for managing threads. It allows tasks to be submitted and assigns them to threads in a pool, handling thread creation, assignment, and cleanup automatically.

**Before getting into creating our Thread Pools and put them into action we should know about Executors class which is also given by Java Concurrency API**

This class contains factory methods for creating different kinds of executor services.

**Let’s see it in action by Creating our first Thread Pool**

Let’s begin with by creating **Single Worker Thread** using ExecutorService & Executors Interfaces

**Example of Single Worker Thread**

package ThreadPool.Executors;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class FixedThreadPoolExample {

public static void main(String[] args) {

// Create a pool of 3 threads

ExecutorService executor = Executors.newSingleThreadExecutor();

// Creating a Runnable Class

Runnable task = new WorkerThread("Task ");

// Submit the task specified by the runnable to the executor service

executor.submit(task);

}

}

class WorkerThread implements Runnable {

private String taskName;

public WorkerThread(String taskName) {

this.taskName = taskName;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName() + " is executing " + taskName);

try {

Thread.sleep(2000); // Simulating work by sleeping for 2 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " finished " + taskName);

}

}

### 

**Output**

pool-1-thread-1 is executing Task

pool-1-thread-1 finished Task

// Here the ExecutorService is still looking for new task

### 

The example above shows how to create an executor service and run a task using it. We use the **Executors.newSingleThreadExecutor()** method to create an ExecutorService that uses one worker thread to execute tasks. If a task is submitted while the thread is busy with another task, the new task will wait in a queue until the thread is free.

If we run this program, we will notice that it never exits because the executor service keeps looking for new tasks until we tell it to shut down.

## **Shutting Down the ExecutorService**

The ExecutorService has two methods for shutting it down:

**shutdown()**: When you call this method on an executor service, it stops accepting new tasks, waits for any tasks that were already submitted to finish, and then shuts down the executor.

**shutdownNow()**: This method interrupts any running tasks and shuts down the executor immediately.

Let’s add shutdown code at the end of our program so that it exits properly.

executor.shutdown(); // Waits for all tasks to complete

// OR

executor.shutdownNow(); // Attempts to stop all tasks immediately

### 

After calling **executor.shutdown()** method

Example of Single Worker Thread with shutdown() method

package ThreadPool.Executors;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class FixedThreadPoolExample {

public static void main(String[] args) {

// Create a pool of 3 threads

ExecutorService executor = Executors.newSingleThreadExecutor();

// Creating a Runnable Class

Runnable task = new WorkerThread("Task ");

// Submit the task specified by the runnable to the executor service

executor.submit(task);

// Shut down the executor

executor.shutdown();

}

}

class WorkerThread implements Runnable {

private String taskName;

public WorkerThread(String taskName) {

this.taskName = taskName;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName() + " is executing " + taskName);

try {

Thread.sleep(2000); // Simulating work by sleeping for 2 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " finished " + taskName);

}

}

### 

**Output**

pool-1-thread-1 is executing Task

pool-1-thread-1 finished Task

PS C:\Users\sukum\Downloads\test> // ExecutorService has been closed

### **Let’s see the Types of Thread Pools**

**Fixed Thread Pool:**

A fixed thread pool has a set number of threads, like a fixed number of cafe workers.

Ideal for tasks with a stable workload.

**Cached Thread Pool**

Imagine a cafe where workers come and go depending on demand. During busy hours, extra workers join in, and they leave when it’s less busy.

Cached thread pools create threads as needed and reuse idle ones, perfect for short, quick tasks.

### 

### **Scheduled Thread Pool**

A **Scheduled Thread Pool** schedules tasks to run after a delay or periodically. It’s great for recurring tasks.

**Analogy:** A maintenance team that inspects equipment at regular intervals. They check on schedule, ensuring everything runs smoothly.

**Fixed Thread Pool**A **Fixed Thread Pool** has a set number of threads. If more tasks are submitted than available threads, tasks are queued until a thread is free.

**Analogy:** Think of it as a call center with a fixed number of agents (threads). If all agents are on calls, new calls must wait until an agent is available.

**Why Use Fixed Thread Pool?**

**Predictable Resource Use:** Limits the number of concurrent tasks.

**Avoids Overload:** Prevents creating too many threads, which could slow down the system.

**Usage:** Useful for tasks with a known, stable workload like processing orders.

**Now let’s create ‘Fixed Thread Pool’** **using ExecutorService & Executors Interfaces**

Using a fixed thread pool is like having a fixed number of workers in a cafe, say three workers. If five customers come in at once, three will be served immediately, while the other two wait until a worker is available.

A **Fixed Thread Pool** is created with a specific number of threads. This is suitable for applications where the workload is expected

**Example of Fixed Thread Pool**

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class FixedThreadPoolExample {

public static void main(String[] args) {

// Create a pool of 3 threads

ExecutorService executor = Executors.newFixedThreadPool(3);

// Submit 5 tasks to the executor

for (int i = 1; i <= 5; i++) {

Runnable task = new WorkerThread("Task " + i);

executor.submit(task);

}

// Shut down the executor

executor.shutdown();

}

}

class WorkerThread implements Runnable {

private String taskName;

public WorkerThread(String taskName) {

this.taskName = taskName;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName() + " is executing " + taskName);

try {

Thread.sleep(2000); // Simulating work by sleeping for 2 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " finished " + taskName);

}

}

### 

**Output**

pool-1-thread-2 is executing Task

pool-1-thread-1 is executing Task

pool-1-thread-3 is executing Task

pool-1-thread-3 finished Task

pool-1-thread-1 finished Task

pool-1-thread-2 finished Task

pool-1-thread-1 is executing Task

pool-1-thread-3 is executing Task

pool-1-thread-3 finished Task

pool-1-thread-1 finished Task

PS C:\Users\sukum\Downloads\test>

### 

**Explanation:**

We create a thread pool with **3 threads** using Executors.newFixedThreadPool(3).

We submit **5 tasks** to this pool. The first 3 tasks will be executed concurrently, while the remaining 2 will wait in the queue.

Once a thread finishes its task, it will pick up a new one from the queue.

After submitting all tasks, we call **shutdown()** to stop accepting new tasks and allow the executor to terminate after completing all submitted tasks.

In the above example, we created an executor service with a fixed thread pool of **3 threads**. A fixed thread pool is a common type of thread pool often used in programs that need multiple tasks to run at the same time.

In a fixed thread pool, the executor service always keeps the same number of threads running. If a thread stops or fails for any reason, the service immediately creates a new thread to take its place.

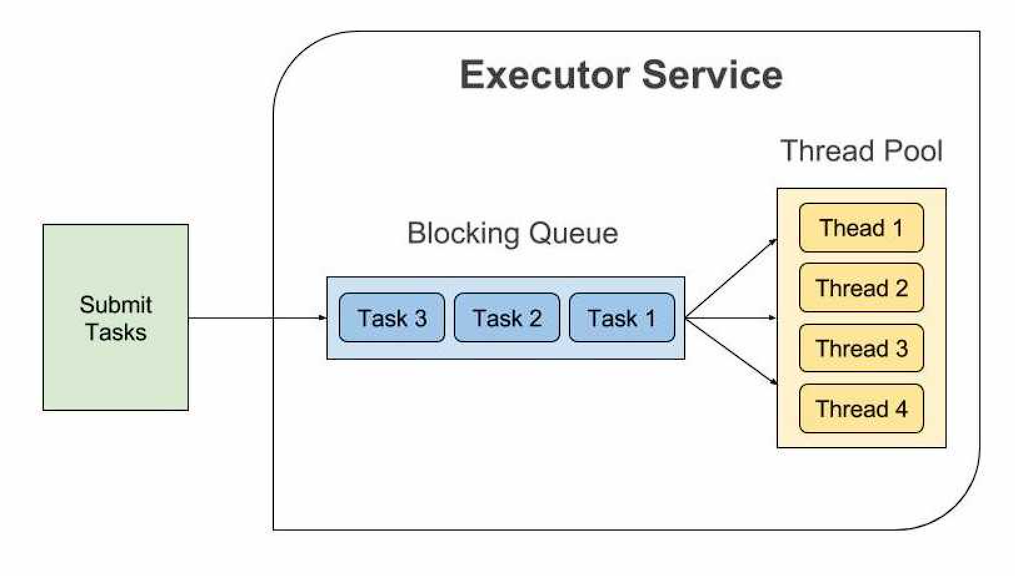
When a new task is submitted, the executor service finds an available thread from the pool and runs the task on that thread. If all threads are busy with other tasks, the new tasks wait in a line **(or queue)** until a thread is free to pick up the next task.

**Blocking Queue**

Creating a thread takes a lot of time and resources, so we should try to create threads only when really needed. Using worker threads helps reduce the time and effort spent creating new threads because the executor service only creates the thread pool once. After that, it can reuse the threads for different tasks.

We already discussed an example of a thread pool called a fixed thread pool.

Tasks are sent to the thread pool through an internal waiting line called a **Blocking Queue**. If there are more tasks than available threads, the extra tasks are placed in this queue to wait until a thread is free. If the queue is full, new tasks will be turned away and won’t be added to the pool.



**Cached Thread Pool**

A **Cached Thread Pool** is a special kind of thread pool in Java that creates new threads when needed and reuses them when they’re free. It’s great for handling tasks that come suddenly in large numbers.

**Analogy**: Think of it like a taxi company. During busy hours, more taxis (threads) are sent out to pick up passengers. But when it’s not busy, only a few taxis are used, and the rest are available for the next busy time.

**Why Use Cached Thread Pool?**

**Handles Sudden High Demand**: Adds more threads when there are lots of tasks, so work can be done quickly.

**Saves Resources**: Reuses threads when there are fewer tasks, so extra threads aren’t sitting idle, saving memory and processing power.

**Usage:** Used in systems with sudden or unpredictable task loads, like chat applications where user activity varies.

**Now let’s create ‘Cached Thread’ Pool** **using ExecutorService & Executors Interfaces**

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class CachedThreadPoolExample {

public static void main(String[] args) {

// Create a cached thread pool

ExecutorService executor = Executors.newCachedThreadPool();

// Submit 5 tasks to the executor

for (int i = 1; i < 5; i++) {

Runnable task = new WorkerThread("Task " + i);

executor.submit(task);

}

// Shut down the executor

executor.shutdown();

}

}

class WorkerThread implements Runnable {

private String taskName;

public WorkerThread(String taskName) {

this.taskName = taskName;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName() + " is executing " + taskName);

try {

Thread.sleep(2000); // Simulating work by sleeping for 2 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " finished " + taskName);

}

}

### 

**Explanation of above code:**

A **Cached Thread Pool** does not have a fixed number of threads. It creates new threads as needed and will reuse idle threads if available.

When tasks are submitted, it either reuses an available thread or creates a new one if all threads are busy.

This pool is ideal when the tasks are small and numerous, or when the workload varies greatly.

**Scheduled Thread Pool**

A **Scheduled Thread Pool** runs tasks after a delay or at fixed intervals. It’s perfect for tasks that need to repeat at regular times.

**Analogy:** Like setting up an automatic coffee maker that brews coffee every morning at the same time.

**Why Use Scheduled Thread Pool?**

**Automates Repetitive Tasks:** Schedules tasks to run at specific intervals.

**Improves Consistency:** Ensures tasks are executed on schedule without manual intervention.

**Usage:** Commonly used for tasks like logging system status or checking server health at regular intervals.

**Example:**

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

public class SimpleSchedulerExample {

public static void main(String[] args) {

// Create a scheduled thread pool with 1 thread

ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

// Schedule a task to run after a 2-second delay

scheduler.schedule(new Task("Delayed Task"), 2, TimeUnit.SECONDS);

// Schedule a task to run every 3 seconds after an initial delay of 1 second

scheduler.scheduleAtFixedRate(new Task("Periodic Task"), 1, 3, TimeUnit.SECONDS);

// Stop the scheduler after 10 seconds

scheduler.schedule(new TaskStopper(scheduler), 10, TimeUnit.SECONDS);

}

}

// A simple task that prints a message

class Task implements Runnable {

private String message;

public Task(String message) {

this.message = message;

}

@Override

public void run() {

System.out.println(message);

}

}

// A task to stop the scheduler

class TaskStopper implements Runnable {

private ScheduledExecutorService scheduler;

public TaskStopper(ScheduledExecutorService scheduler) {

this.scheduler = scheduler;

}

@Override

public void run() {

scheduler.shutdown();

System.out.println("Scheduler stopped.");

}

}

### **Explanation of This Code:**

### **Focus on Scheduling**:

### The Task class is a simple Runnable that prints a message when executed.

### The TaskStopper class gracefully shuts down the scheduler.

### **Smaller Example**:

### Uses only one thread in the scheduler for simplicity.

### Highlights both schedule and scheduleAtFixedRate methods.

### **Output Example**:

### **After 2 seconds**: "Delayed Task" is printed.

### **Every 3 seconds after a 1-second delay**: "Periodic Task" is printed.

### **After 10 seconds**: "Scheduler stopped." is printed, and the program exits.

### 

### 

## **Callable and Future Interfaces**

**Callable** and **Future** are tools in Java used for **asynchronous programming**.

They allow us to:

* Run tasks in the background.
* Get the result of the task when it’s done.
* Check if the task is still running or finished.
* Handle exceptions if something goes wrong during the task.

### 

And not only this let’s see **Why Were Callable and Future were Introduced ?**

Before **Callable** was introduced, we used the **Runnable** interface to run tasks in threads. However:

* **Runnable** could not return a result.
* **Runnable** could not throw checked exceptions.

To solve these issues, Java introduced:

* **Callable**, which can return a result and throw exceptions.
* **Future**, which is used to get the result of a Callable once it finishes.

Lets see the **Real-World Analogy**

**Callable and Future** are like ordering food at a restaurant:

* You place your order (**submit a Callable task**).
* The waiter gives you a token (**Future**) to track your order.
* You can continue doing other things while the food is being prepared (**asynchronous execution**).
* When the food is ready, you hand over the token to get your meal (**get() method in Future**).
* If the restaurant is taking too long, you can check on your order (**isDone() method**) or even cancel it (**cancel() method**).

Now let’s see **How Callable Works**

The **Callable** interface is similar to **Runnable** but with two key differences:

1. It can return a value (using the call() method).
2. It can throw checked exceptions.

**Code Example:**

import java.util.concurrent.Callable;

/\*\*

\* This class demonstrates a Callable task that calculates the square of a given number.

\* Callable is a functional interface in Java that can return a result and throw exceptions.

\*/

public class MyCallableTask implements Callable<Integer> {

// The number whose square will be calculated

private int number;

/\*\*

\* Constructor to initialize the number for the task.

\*

\* @param number the number to calculate the square of

\*/

public MyCallableTask(int number) {

this.number = number;

}

/\*\*

\* The main logic of the Callable task.

\* This method will be executed when the task is submitted to an ExecutorService.

\*

\* @return the square of the number

\* @throws Exception if any error occurs during execution

\*/

@Override

public Integer call() throws Exception {

// Calculate the square of the number

int result = number \* number;

// Print a message indicating the calculation

System.out.println("Calculating square of " + number + ": " + result);

// Return the result

return result;

}

}

### **Explanation of this code**

**call() Method:**

* This method contains the logic for the task. It calculates the square of the number, prints a message, and then returns the result.
* The call() method is similar to the run() method in Runnable, but it can return a value and throw exceptions.

**Output Example:**

* When executed, the program will print the calculation process and return the result:

Calculating square of 5: 25

### 

### Now let’s see **How Future Works and supports Callable**

The **Future** interface is used to interact with the result of a **Callable** task. It provides methods to:

* **get()**: Retrieve the result once the task is done.
* **isDone()**: Check if the task is complete.
* **cancel()**: Cancel the task if it hasn’t started yet.

**Code Example:**

import java.util.concurrent.\*;

public class CallableAndFutureExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newSingleThreadExecutor();

// Submit a Callable task

Future<Integer> future = executor.submit(new MyCallableTask(5));

try {

// Check if the task is done

if (!future.isDone()) {

System.out.println("Task is still running...");

}

// Get the result of the task (blocks until result is ready)

Integer result = future.get();

System.out.println("Result of task: " + result);

} catch (Exception e) {

e.printStackTrace();

} finally {

executor.shutdown();

}

}

}

### 

### **Advantages of Callable and Future**

* **Asynchronous Execution**: Allows tasks to run in the background without blocking the main program.
* **Result Handling**: Returns results of tasks and handles exceptions.
* **Thread Pool Integration**: Works seamlessly with Java's ExecutorService for better thread management.