

# ExecutorService in Java

java multithreading executor service explain in detail  
cover all concepts with code

Sure! Let's dive deep into **Java Multithreading using ExecutorService**, covering concepts, advantages, how it works, and with detailed code examples.

---

## Java Multithreading: ExecutorService Explained in Detail

---

### 1. Introduction to Multithreading in Java

- **Multithreading** allows concurrent execution of two or more parts of a program to maximize CPU utilization.
  - Threads can be created by extending `Thread` class or implementing `Runnable`.
  - But managing threads manually is error-prone and complicated.
  - **ExecutorService** framework (part of `java.util.concurrent` package) provides a high-level API for managing a pool of threads efficiently.
- 

### 2. What is ExecutorService?

- `ExecutorService` is an interface representing a thread pool.
  - It **abstracts thread creation and management** from the developer.
  - It **executes submitted tasks asynchronously** using a pool of reusable threads.
  - It handles thread lifecycle, scheduling, and queuing of tasks internally.
- 

### 3. Why use ExecutorService?

- **Thread reuse:** No overhead of creating new thread every time.
- **Task scheduling:** Can schedule and execute tasks in background.
- **Thread pool management:** Control max threads and system resources.
- **Graceful shutdown:** Ability to shutdown thread pool cleanly.
- **Future & Callable support:** Return result from threads asynchronously.

---

## 4. Core Interfaces and Classes

Interface/Class	Description
Executor	Simple interface with <code>execute(Runnable)</code>
ExecutorService	Extends Executor with lifecycle management
ThreadPoolExecutor	Core implementation of ExecutorService
ScheduledExecutorService	Extends ExecutorService for scheduling tasks

---

## 5. ExecutorService Methods Overview

Method	Description
<code>submit(Runnable/Callable)</code>	Submits task and returns a Future
<code>execute(Runnable)</code>	Executes a task without returning Future
<code>shutdown()</code>	Initiates orderly shutdown, no new tasks
<code>shutdownNow()</code>	Attempts to stop all running tasks
<code>awaitTermination(timeout, unit)</code>	Waits for termination after shutdown
<code>invokeAll(Collection&lt;Callable&gt;)</code>	Executes all tasks and waits for results

---

## 6. How to Create ExecutorService?

Common factory methods in `Executors` utility class:

Method	Description
<code>Executors.newFixedThreadPool(n)</code>	Fixed number of threads in pool
<code>Executors.newCachedThreadPool()</code>	Creates threads as needed, reuses idle threads
<code>Executors.newSingleThreadExecutor()</code>	Single thread executor
<code>Executors.newScheduledThreadPool(n)</code>	Thread pool for scheduled/delayed tasks

## 7. Runnable vs Callable

Aspect	Runnable	Callable
Return value	No return value	Returns a value
Exception	Cannot throw checked exceptions	Can throw checked exceptions
Used with	<code>execute()</code> , <code>submit()</code>	<code>submit()</code>

## 8. Code Examples

### Example 1: Using ExecutorService with Runnable

```
java
```

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceExample {

    public static void main(String[] args) {
        // Create a fixed thread pool with 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Runnable task
        Runnable task = () -> {
            String threadName = Thread.currentThread().getName();
            System.out.println("Running task in " + threadName);
            try {
                Thread.sleep(1000); // simulate work
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        };

        // Submit 5 tasks for execution
        for (int i = 0; i < 5; i++) {
            executor.execute(task);
        }

        // Shutdown executor
        executor.shutdown();
    }
}

```

## Output:

cpp

```

Running task in pool-1-thread-1
Running task in pool-1-thread-2
Running task in pool-1-thread-3
Running task in pool-1-thread-1
Running task in pool-1-thread-2

```

## Example 2: Using ExecutorService with Callable and Future

```
java

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableExample {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Callable<String> callableTask = () -> {
            Thread.sleep(500);
            return "Result from " + Thread.currentThread().getName();
        };

        Future<String> future1 = executor.submit(callableTask);
        Future<String> future2 = executor.submit(callableTask);

        System.out.println(future1.get()); // Blocks until result is available
        System.out.println(future2.get());

        executor.shutdown();
    }
}
```

### Output:

```
sql

Result from pool-1-thread-1
Result from pool-1-thread-2
```

---

## Example 3: Shutdown and AwaitTermination

```
java
```

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ShutdownExample {

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Runnable task = () -> {
            try {
                System.out.println("Task started in " + Thread.currentThread().getName());
                Thread.sleep(2000);
                System.out.println("Task finished in " + Thread.currentThread().getName());
            } catch (InterruptedException e) {
                System.out.println("Task interrupted");
                Thread.currentThread().interrupt();
            }
        };

        executor.submit(task);
        executor.submit(task);

        executor.shutdown(); // No new tasks accepted

        // Wait for termination for max 5 seconds
        if (executor.awaitTermination(5, TimeUnit.SECONDS)) {
            System.out.println("All tasks completed.");
        } else {
            System.out.println("Timeout reached, forcing shutdown.");
            executor.shutdownNow(); // Cancel running tasks
        }
    }
}

```

## 9. Important Notes

- Use **fixed thread pool** when you want a predictable number of threads.
- Use **cached thread pool** for short-lived asynchronous tasks.
- Always call **shutdown()** to free resources after tasks are done.

- Avoid submitting blocking tasks in fixed thread pools to prevent thread starvation.
- `Future.get()` blocks until the task completes.
- You can check if executor is shutdown with `isShutdown()` or terminated with `isTerminated()`.

---

## 10. Summary

Feature	Description
<code>ExecutorService</code>	Manages thread pools & task execution
Thread Pools	Efficient reuse of fixed or dynamic threads
Runnable vs Callable	Callable supports return & exceptions
Task Submission Methods	<code>execute()</code> , <code>submit()</code> , <code>invokeAll()</code>
Lifecycle Management	<code>shutdown()</code> , <code>shutdownNow()</code> , <code>awaitTermination()</code>
Futures	Obtain async results

---

If you want, I can also show you examples with `ScheduledExecutorService` for scheduled tasks or custom `ThreadPoolExecutor` usage!

---

Feel free to ask if you want me to explain any specific part in more detail or provide more examples!

thats it there is so many checks and ways to get data from Future right?

Exactly! When working with `Future` in Java's `ExecutorService`, there are multiple ways and nuances to handle task results and check their status. Let me break down the key points and options for you:

---

## Handling and Getting Data from `Future`

---

### What is `Future` ?

- Represents the result of an asynchronous computation.
  - Provides methods to check if the task is complete, to wait for completion, and to retrieve the result.
- 

### Common `Future` Methods to Retrieve Data or Status

Method	Description
<code>get()</code>	Waits (blocks) until the task completes and returns result. Throws exception if task failed.
<code>get(long timeout, TimeUnit unit)</code>	Waits only for specified timeout. Throws <code>TimeoutException</code> if timeout occurs before completion.
<code>isDone()</code>	Returns true if task completed (normally or exceptionally).
<code>isCancelled()</code>	Returns true if task was cancelled before completion.
<code>cancel(boolean mayInterruptIfRunning)</code>	Attempts to cancel the task. If running, may interrupt it.

---

### Ways to Use `Future` to Get Data Safely



## 1. Simple blocking get:

java

```
Future<String> future = executor.submit(callableTask);  
String result = future.get(); // Waits indefinitely until done
```

- Simple, but blocks current thread.
  - Not always ideal if you want responsiveness or timeout control.
- 

## 2. Timed get with timeout:

java

```
try {  
    String result = future.get(2, TimeUnit.SECONDS);  
} catch (TimeoutException e) {  
    System.out.println("Task timed out");  
}
```

- Prevents blocking forever.
  - You can handle timeout scenarios gracefully.
- 

## 3. Polling with `isDone()`:

java

```
while (!future.isDone()) {  
    System.out.println("Waiting for result...");  
    Thread.sleep(100); // Sleep before next check  
}  
String result = future.get(); // Now it won't block
```

- Useful if you want to do other work while waiting.
  - Avoids blocking the main thread unnecessarily.
- 

## 4. Check if cancelled or cancel task:

java

```
if (!future.isDone()) {  
    future.cancel(true); // Interrupt if running  
}  
if (future.isCancelled()) {  
    System.out.println("Task was cancelled");  
}
```

- Cancel long-running or stuck tasks.
- Be mindful of thread interruption and task design.

---

## 5. Combine polling + timeout:

```
java  
  
long startTime = System.currentTimeMillis();  
long timeout = 5000; // 5 seconds  
while (!future.isDone()) {  
    if (System.currentTimeMillis() - startTime > timeout) {  
        future.cancel(true);  
        System.out.println("Timeout - cancelled task");  
        break;  
    }  
    Thread.sleep(100);  
}  
if (!future.isCancelled()) {  
    System.out.println("Result: " + future.get());  
}
```

---

## Handling Exceptions When Getting Result

- `get()` can throw:
  - `ExecutionException` if the task threw an exception.
  - `InterruptedException` if current thread is interrupted.
  - `TimeoutException` if timeout reached in timed get.

Always handle these exceptions when calling `get()`.

---

## Summary: Which to Use When?

Scenario	Recommended approach
Wait indefinitely for result	<code>future.get()</code>
Wait with maximum time	<code>future.get(timeout, unit)</code>
Do other work and check periodically	<code>isDone()</code> with polling
Cancel long-running task	<code>cancel()</code>
Handle possible task exceptions	Wrap <code>get()</code> in try-catch

If you want, I can provide example code snippets demonstrating each of these patterns. Would you like that?

implement and spring boot and executor service, give example from ssearch

Got it! Here's a complete example showing how to integrate **ExecutorService** in a **Spring Boot** application, implementing an async search service that performs search tasks concurrently.

## Spring Boot + ExecutorService Example: Async Search Service

### Overview

- Spring Boot REST API endpoint `/search?query=...`

- On each request, it submits the search task to an ExecutorService thread pool
  - Returns search results asynchronously after task completion
  - Uses `ExecutorService` directly (not Spring's `@Async` for demonstration)
  - Graceful shutdown of thread pool on app stop
- 

## Step 1: Create Spring Boot Project

If you want, you can create a Spring Boot project using [start.spring.io](https://start.spring.io) with dependencies:

- Spring Web
- 

## Step 2: Add ExecutorService Configuration

Create a configuration class to instantiate a fixed thread pool ExecutorService bean:

```
java

package com.example.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

@Configuration
public class ExecutorConfig {

    @Bean(destroyMethod = "shutdown")
    public ExecutorService taskExecutor() {
        // Fixed thread pool of size 4
        return Executors.newFixedThreadPool(4);
    }
}
```

---

## Step 3: Create SearchService Using ExecutorService

This service submits search tasks to the ExecutorService and returns Future results.

java

```
package com.example.demo.service;

import org.springframework.stereotype.Service;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;

@Service
public class SearchService {

    private final ExecutorService executorService;

    public SearchService(ExecutorService executorService) {
        this.executorService = executorService;
    }

    public Future<String> searchAsync(String query) {
        Callable<String> searchTask = () -> {
            // Simulate time-consuming search operation
            Thread.sleep(2000);
            // Return mock result
            return "Results for query: " + query;
        };

        return executorService.submit(searchTask);
    }
}
```

---

## Step 4: Create REST Controller to Expose Search Endpoint

The controller accepts a query param, calls service, waits for result and returns it.

java

```
package com.example.demo.controller;

import com.example.demo.service.SearchService;
import org.springframework.web.bind.annotation.GetMapping;
```

```

import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

@RestController
public class SearchController {

    private final SearchService searchService;

    public SearchController(SearchService searchService) {
        this.searchService = searchService;
    }

    @GetMapping("/search")
    public String search(@RequestParam String query) {
        Future<String> futureResult = searchService.searchAsync(query);

        try {
            // Wait for result (blocking)
            return futureResult.get();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return "Search interrupted";
        } catch (ExecutionException e) {
            return "Error occurred during search: " + e.getCause().getMessage();
        }
    }
}

```

## Step 5: Run and Test

Run the Spring Boot app ( `DemoApplication.java` ) and call:

```
bash
```

```
GET http://localhost:8080/search?query=springboot
```

You will see the response after approx 2 seconds:

rust

Results **for query**: springboot

---

## Optional: Non-blocking Controller (Return Future Directly)

If you want to return the `Future` itself for asynchronous handling, you can return `Callable<String>` from the controller:

java

```
@GetMapping("/search-async")
public Callable<String> searchAsync(@RequestParam String query) {
    return () -> searchService.searchAsync(query).get();
}
```

Spring MVC will execute it asynchronously.

---

## Summary

- Created a Spring-managed `ExecutorService` thread pool bean.
- Injected it into a service that submits search tasks.
- REST controller calls service and returns the result after waiting on `Future`.
- Graceful shutdown ensured by Spring destroying the `ExecutorService` bean.

---

If you want me to extend this with **`ScheduledExecutorService`**, custom **`ThreadPoolExecutor`**, or integration with Spring's `@Async`, just ask!

complete differences between `Runnable` and `Callable`  
, if required write code also

Sure! Here's a **complete comparison** between **`Runnable`** and **`Callable`** in Java, with explanations, key differences, and code examples.

---

## Runnable vs Callable in Java

Feature	Runnable	Callable
Package	<code>java.lang</code>	<code>java.util.concurrent</code>
Return Type	Does <b>not</b> return a result ( <code>void run()</code> )	Returns a result ( <code>call()</code> ) returns <code>v</code> )
Exception Handling	Cannot throw checked exceptions	Can throw checked exceptions
Method	<code>void run()</code>	<code>V call()</code>
Used in Executor	Used with <code>execute()</code> and <code>submit()</code> (but returns <code>Future&lt;?&gt;</code> )	Used with <code>submit()</code> , returns <code>Future&lt;V&gt;</code> with result
Supports Generics	No	Yes, generic return type
Result Retrieval	No direct result; must use other ways (e.g., shared variables)	Can get result via <code>Future.get()</code>
Introduced in	Java 1.0 (core Java)	Java 5 (concurrency utilities)

## Runnable: Details and Example

- Defines a task to run, no return value.
- You cannot throw checked exceptions from `run()` .
- Suitable for fire-and-forget tasks.

## Runnable example

java

```
public class RunnableExample implements Runnable {
    @Override
    public void run() {
        System.out.println("Runnable running in " + Thread.currentThread().getName());
        // no return value
    }

    public static void main(String[] args) {
```



```
RunnableExample task = new RunnableExample();
Thread thread = new Thread(task);
thread.start();
}
}
```

---

## Callable: Details and Example

- Represents a task that returns a value and can throw exceptions.
- `call()` method returns a generic type.
- Can be submitted to `ExecutorService` and get a `Future<V>` to retrieve result.

### Callable example

```
java

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableExample implements Callable<String> {

    @Override
    public String call() throws Exception {
        return "Callable executed in " + Thread.currentThread().getName();
    }

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        CallableExample task = new CallableExample();
        Future<String> future = executor.submit(task);
        String result = future.get(); // blocking call
        System.out.println(result);
        executor.shutdown();
    }
}
```

## Summary Table

Aspect	Runnable	Callable
Return value	No (void)	Yes (generic type <code>V</code> )
Exception handling	Cannot throw checked exceptions	Can throw checked exceptions
Result retrieval	No direct support	Via <code>Future.get()</code>
Supports Generics	No	Yes
Introduced	Java 1.0	Java 5 (concurrency framework)
Usage in Executor	<code>execute()</code> or <code>submit()</code>	<code>submit()</code> only
Suitable for	Simple tasks, fire-and-forget	Tasks needing result or exception

If you want, I can also show you an example of converting a Runnable to Callable or handling exceptions in Callable. Just ask!

`thread.start();` will start thread execution, then what about executor service when it will start execution

Great question! Let's clarify **when and how ExecutorService starts executing tasks compared to** `thread.start()`.

## Thread `start()` vs ExecutorService Task Execution

### 1. `thread.start()`

- When you create a `Thread` object and call `start()`, the JVM immediately schedules that thread for execution.

- The thread transitions from the NEW state to RUNNABLE state.
  - The OS thread scheduler picks it up quickly (usually immediately or very soon).
  - So, `start()` **directly triggers the thread to run its `run()` method** asynchronously.
- 

## 2. ExecutorService Task Submission

- When you submit a `Runnable` or `Callable` task to an `ExecutorService` (using `submit()` or `execute()`), **the task is added to an internal task queue.**
  - The actual execution depends on the type of `ExecutorService` and its thread pool:
    - **If there are idle threads in the pool**, one will pick the task immediately and start running it.
    - **If all threads are busy**, the task will wait in the queue until a thread is free.
  - So, **task execution is not immediate upon submission; it depends on thread availability and queueing.**
  - This design enables thread reuse and controlled concurrency.
- 

## In Short

Operation	When execution starts
<code>thread.start()</code>	Immediately schedules and runs the thread (as soon as possible)
<code>executor.submit(task)</code>	Adds task to queue; runs only when a thread from pool is free

---

## Visual Example

text

ThreadPool size = 2

submit Task A -> picked immediately by Thread-1 -> runs

submit Task B -> picked immediately by Thread-2 -> runs

submit Task C -> queued, waits until Thread-1 or Thread-2 finishes Task A or B

---

## Why `ExecutorService`'s delayed start is good:

- Avoids overhead of creating new thread per task
- Controls max concurrent threads
- Efficient for large number of short-lived tasks
- Avoids resource exhaustion by limiting thread count

---

If you want, I can show you code demonstrating task queueing and execution delay in `ExecutorService`. Would you like that?