ExecutorService

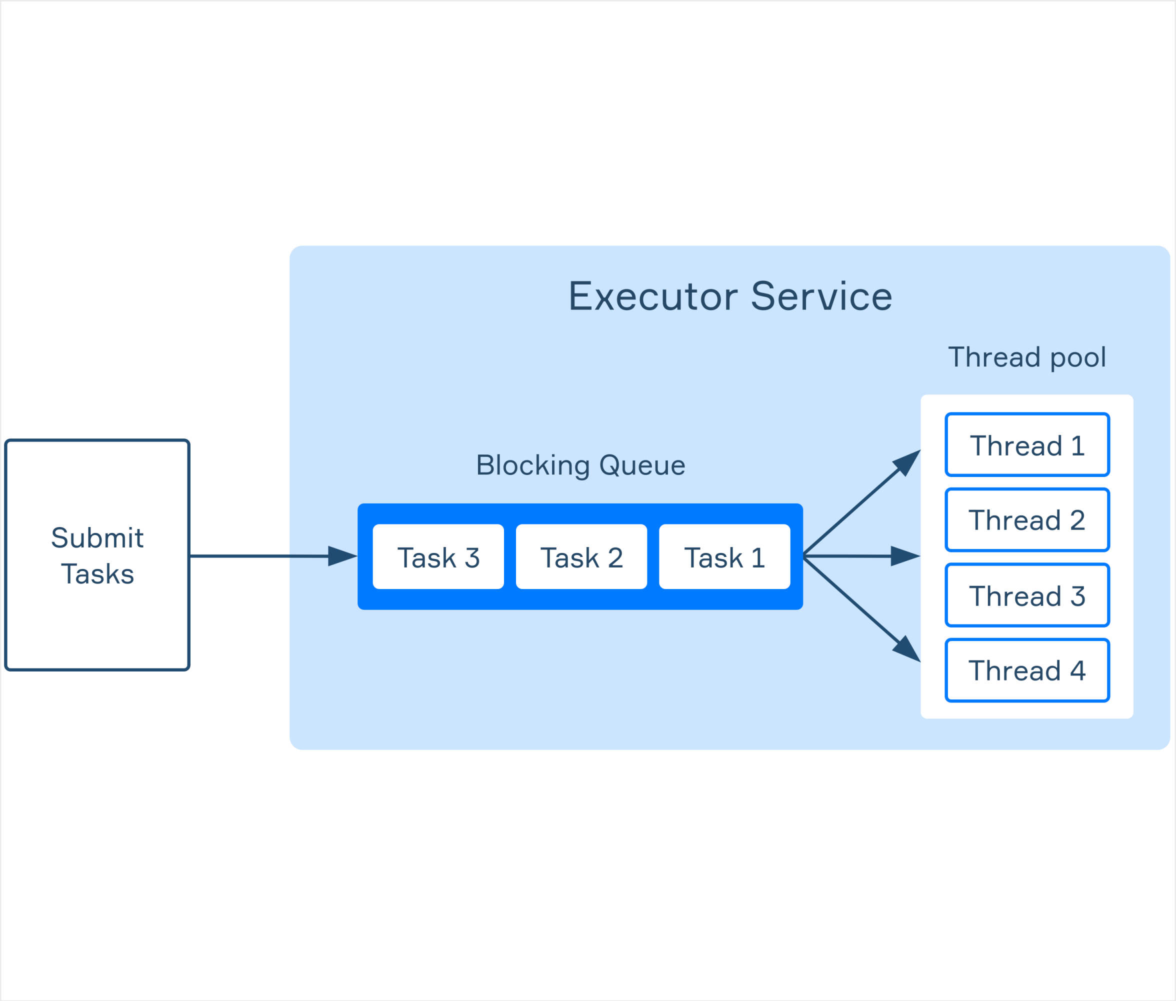
The main purpose of using multithreading is concurrent execution and multi-user support. In the case of multi-core processors, we even have parallel execution of tasks.

Another advantage of multithreading is that thread creation consumes fewer resources than process creation.

The concurrent package for managing threads includes a facility called **executor services**.

Executor services simplify the asynchronous execution of tasks and automatically provide a pool of threads and an interface for assigning tasks to them.

They facilitate task execution by ensuring we do not have to deal with the Thread class. Tasks are represented as Runnable or Callable objects. You can execute both types of tasks in a single thread or thread pools with the help of an ExecutorService.



First, tasks are submitted to an ExecutorService. After submission, they are placed in a queue. Available threads pick up the tasks from the queue and execute them.

When you start tasks using an Executor of the java.util.concurrent package, it is not necessary to resort to the low-level threading functionality of the Thread class.

It's enough to create an object of ExecutorService type with the required properties and pass a task of Callable type to it for execution.

Initialization of an ExecutorService

As ExecutorService is an interface, any of its implementations can be used. There are several implementations to choose from in the java.util.concurrent package or you can create your own. The main task is to execute tasks in several threads.
A thread pool with a fixed number of threads can be obtained by calling the static newFixedThreadPool() method of the Executors class:
ExecutorService service = Executors.newFixedThreadPool(2);

Here, only two threads will be active to process tasks. If more than two tasks are submitted, they are held in a queue until threads become available.

All threads exist until the pool is closed.
A pool of cached threads can be obtained by calling the
static newCachedThreadPool() method of the Executors class:

```
ExecutorService executor = Executors.newCachedThreadPool();
```

The newCachedThreadPool() method creates a caching thread pool that creates threads as needed but reuses idle threads and cleans up threads that have been idle for a while.

A cached thread executor is suitable for applications that run many short tasks.

The newCachedThreadPool() method creates threads automatically, and there is no limit on their number. This can lead to unlimited thread growth. Additionally, creating and deleting threads consumes resources. So, despite the seeming convenience, it must be used carefully.

Sometimes you need to execute code after some time. In such cases, the ScheduledExecutorService class comes to the rescue.

It allows you to set the code to be executed in one or more threads and configure the delay of execution.

The delay can be the time between two successive runs or between the end of one run and the start of another.

The methods of a ScheduledExecutorService return a ScheduledFuture object that also contains a deferral value for executing ScheduledFuture.

```
ScheduledExecutorService service =
Executors.newSingleThreadScheduledExecutor();
service.schedule(() -> System.out.println("Executed"), 5, TimeUnit.SECONDS);
```

If you want to execute tasks with a **zero initial delay** and **every second** regardless of the completion time of the previous tasks, do this:

```
ScheduledExecutorService service =
Executors.newSingleThreadScheduledExecutor();

service.scheduleAtFixedRate(() -> System.out.println(
      "Executed every second"), 0, 1, TimeUnit.SECONDS);
```

Finally, if you want to schedule code execution with **zero initial delays** and **one-second intervals** between executions, do the following. Each task will start after the completion of the previous task:

```
ScheduledExecutorService service =
Executors.newSingleThreadScheduledExecutor();

service.scheduleWithFixedDelay(() -> System.out.println(
       "Executed after a one-second interval"), 0, 1, TimeUnit.SECONDS);
```

The standard thread naming scheme is **pool-N-thread-M**, where N is the pool's serial number (every time you create a new pool, the global counter N is incremented), and M is the thread's serial number in the pool.
The ThreadFactory class can help to create threads with a specific name.

```
public class ThreadFactorySimple {
   public static void main(String[] args) {
      ThreadFactory threadFactory = new ThreadFactory() {
         private int counter = 0;

         @Override
         public Thread newThread(Runnable runnable) {
            return new Thread(runnable, "thread-pool-name " + counter++);
            // thread-pool-name 1
            // thread-pool-name 2
            // etc.
         }
      };

      for (int i = 0; i < 3; i++) {

System.out.println(threadFactory.newThread(ThreadFactorySimple::run).getNa
me());
      }
   }

   private static void run() {
      // Some logic
   }
}
```

With the Google package com.google.common.util.concurrent.ThreadFactoryBuilder, you can set the thread name more simply:

```
public class ThreadFactorySimple {
   public static void main(String[] args) {
      ThreadFactory threadFactory = new ThreadFactory() {
         private final AtomicInteger counter = new AtomicInteger(0);
```

```
            @Override
            public Thread newThread(Runnable runnable) {
                // Create a new thread with a name, e.g., thread-pool-name-1
                return new Thread(
                    runnable, "thread-pool-name-" + counter.getAndIncrement());
            }
        };
        ExecutorService service = Executors.newFixedThreadPool(3,
threadFactory);
        for (int i = 0; i < 3; i++) {
            service.submit(
                    () -> System.out.println(
                        Thread.currentThread().getName() + " is running"));
        }
        service.shutdown();
    }
}
```

With the Google package com.google.common.util.concurrent.ThreadFactoryBuilder, you can set the thread name more simply:

```
public class ThreadFactorySimple {
    public static void main(String[] args) {
        ThreadFactory threadFactory = new ThreadFactory() {
            private final AtomicInteger counter = new AtomicInteger(0);

            @Override
            public Thread newThread(Runnable runnable) {
                // Create a new thread with a name, e.g., thread-pool-name-1
                return new Thread(
                    runnable, "thread-pool-name-" + counter.getAndIncrement());
            }
        };

        ExecutorService service = Executors.newFixedThreadPool(3,
threadFactory);
        for (int i = 0; i < 3; i++) {
            service.submit(
                    () -> System.out.println(
                        Thread.currentThread().getName() + " is running"));
        }
        service.shutdown();
    }
}
```

To use this library, you have to include it in your project. This is an example for Gradle:

```
// https://mvnrepository.com/artifact/com.google.guava/guava
```

```
implementation group: 'com.google.guava', name: 'guava', version: '31.1-jre'
```

Variations of ExecutorService's submit and invoke methods

We can assign tasks to an ExecutorService using several methods, including execute(), which is inherited from the Executor interface, and submit(), invokeAny() and invokeAll().
The execute() method is **void** and doesn't allow getting a task's result or checking a task's status (if it's running or not). It takes a Runnable object and executes it asynchronously.

```
public class RunnableExecuteExample {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        executorService.execute(() -> {
            System.out.println("Runnable asynchronous task"); // Runnable
asynchronous task
        });

        executorService.shutdown();
    }
}
```

The submit() and invokeAll() methods return an object or a collection of objects of the Future type. As such, we can get the result of a task or check its status (if it's running or not).

The submit() method queues a task for execution. It takes an object of Callable or Runnable type as an input parameter and returns a parameterized object of the Future type. Using the Future object, you can determine whether a task is finished (using the isDone() method) or access its result (using the get() method) or an exception if an error occurred during the task execution.

```
public class RunnableSubmitExample {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Future<?> future = executorService.submit(() -> {
            System.out.println("Runnable task"); // Runnable task
        });

        System.out.println("Runnable result: " + future.get()); // Runnable result:
null
        // Returns null if the task is finished correctly

        executorService.shutdown();
    }
}
```

The submit(Callable) method of ExecutorService is similar
to submit(Runnable) but accepts a Callable instead of a Runnable.

```
public class SubmitCallableExample {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Future<String> future = executorService.submit(() -> {
            System.out.println("Asynchronous callable");  // Asynchronous callable
            return "Callable Result";
        });

        System.out.println("Future: " + future.get());  // Future: Callable Result

        executorService.shutdown();
    }
}
```

The invokeAny() method accepts a collection of callable objects. Calling this method does not return a Future object but returns the result of one of the called objects (if there was a successful execution):

```
public class InvokeExample {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Set<Callable<String>> callables = new HashSet<>();
        callables.add(() -> "First task");
        callables.add(() -> "Second task");

        String result = executorService.invokeAny(callables);
        System.out.println("Result: " + result); // Result:  First task

        executorService.shutdown();
    }
}
```

The invokeAll() method invokes all Callable objects passed as parameters. It assigns a collection of tasks to an ExecutorService, causing each to run. Results are returned in the form of a list of Future objects, which can be used to get the results of the execution of each called object.

```
public class InvokeAllExample {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Set<Callable<String>> callables = new HashSet<>();
        callables.add(() -> "First task");
        callables.add(() -> "Second task");

        List<Future<String>> futures = executorService.invokeAll(callables);
        for (Future<String> future : futures) {
            System.out.println("Future: " + future.get()); // Future: First task
                                                           // Future: Second task
        }

        executorService.shutdown();
    }
}
```

Termination with the shutdown and shutdownNow methods

Now let's learn how to finish an ExecutorService process.
In some cases, ExecutorServices are very helpful, such as when an app needs to process tasks that appear irregularly or when the number of tasks is not known at compile time. On the other hand, an app could reach its end but not be stopped because a waiting ExecutorService will cause the JVM to keep running. To properly shut down an ExecutorService, we have the shutdown()and shutdownNow() APIs.

The ExecutorService will not be automatically destroyed when there is no task to process. It will stay alive and wait for new work to appear. You have to stop the ExecutorService object because threads in an ExecutorService object don't stop themselves.

The shutdown() method doesn't cause immediate destruction of the ExecutorService. It will make the ExecutorService stop accepting new tasks and shut down after all running threads finish their current work:

```
executorService.shutdown();
```

The shutdownNow() method tries to destroy the ExecutorService immediately and returns a list of tasks that were not taken from the queue. However, this method doesn't guarantee that all the running threads will be stopped at the same time:

```
executorService.shutDownNow();
```

Finally, let's discuss termination with the awaitTermination() method. This method blocks the thread calling it until the ExecutorService has shut down or a given time-out occurs.
Here is an example of calling the ExecutorService awaitTermination() method:

```
public class AwaitTerminationExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 3; i++) {
            executor.execute(() ->
                    System.out.println("Thread running in: " +
Thread.currentThread()));
            // Thread running in: Thread[pool-1-thread-3,5,main]
            // Thread running in: Thread[pool-1-thread-2,5,main]
            // Thread running in: Thread[pool-1-thread-1,5,main]
        }

        // Prevent new tasks from being submitted
        executor.shutdown();

        try {
            // Wait 10 seconds for the tasks to terminate
            if (!executor.awaitTermination(10, SECONDS)) {
                // Cancel currently executing tasks
                executor.shutdownNow();

                // Wait 60 seconds for tasks to respond
                if (!executor.awaitTermination(60, SECONDS)) {
                    System.err.println("Pool did not terminate");
                }
            }
        } catch (InterruptedException ex) {
            // Cancel if the current thread was interrupted
            executor.shutdownNow();
            // Preserve the interrupt status
            Thread.currentThread().interrupt();
        }

    }
}
```

The awaitTermination() method is typically called after calling shutdown() or shutdownNow().