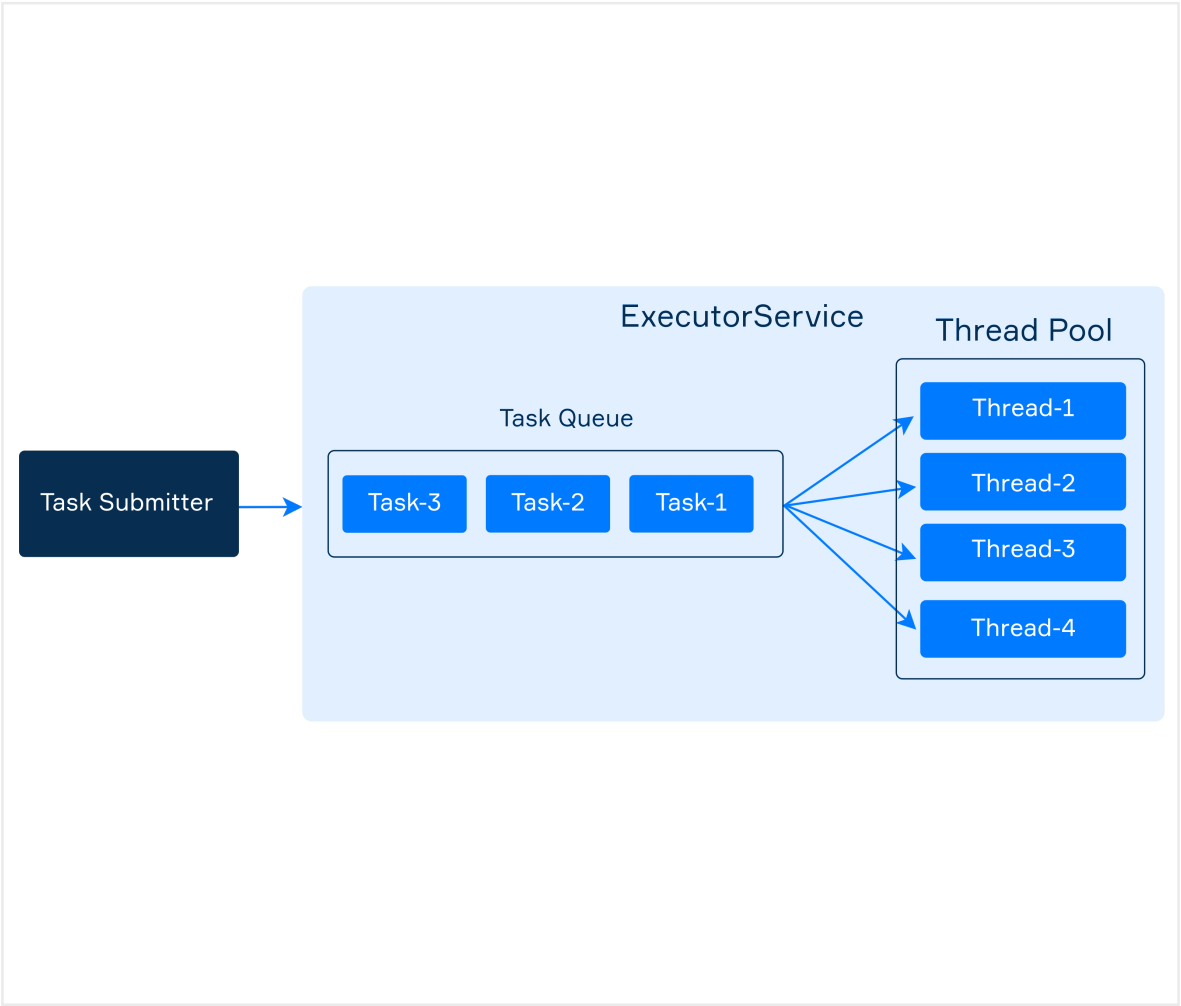
Executors

While it is easy to create several threads and start them, it becomes a problem when your application has hundreds or even thousands of threads running concurrently.

In addition, Thread is a quite low-level class and mixing it with the high-level code of your application may lead to unreadable code and poor architecture in the future. It may also produce some well-known errors such as invoking run() instead of start().

Tasks and executors

To simplify the development of multi-threaded applications, Java provides an abstraction called ExecutorService (or simply **executor**).



It encapsulates one or more threads into a single pool and puts submitted tasks in an internal queue to execute them by using the threads.

This approach clearly isolates tasks from threads and allows you to focus on tasks.

You do not need to worry about creating and managing threads because the executor does it for you.

Creating executors

All types of executors are located in the java.util.concurrent package.

This package also contains a convenient utility class Executors for creating different types of ExecutorServices.

let's create an executor with exactly four threads in the pool:
ExecutorService executor = Executors.newFixedThreadPool(4);

It can execute multiple tasks concurrently and speed up your program by performing somewhat parallel computations.

If one of the threads dies, the executor creates a new one.

Submitting tasks

An executor has the submit method that accepts a Runnable task to be executed.

Since Runnable is a functional interface, it is possible to use a lambda expression as a task.

As an example, here we submit a task that prints **"Hello!"** to the standard output.
executor.submit(() -> System.out.println("Hello!"));

we can declare a class that implements Runnable for our task, and then submit an object of this class. But it is very convenient to use lambda expressions together with **executors** for short tasks.

After invoking submit, the current thread does not wait for the task to complete. It just adds the task to the executor's internal queue to be executed asynchronously by one of the threads.

Stopping executors

An executor continues to work after the completion of a task since threads in the pool are waiting for new coming tasks.

Your program will never stop while at least one executor still works.

There are two methods for stopping executors:

- void shutdown() waits until all running tasks are completed and prohibits submitting of new tasks;
- List<Runnable> shutdownNow() immediately stops all running tasks and returns a list of the tasks that were awaiting execution.

Note that shutdown() does not block the current thread unlike join() of Thread.

If you need to wait until the execution is complete, you can invoke awaitTermination(...) with the specified waiting time.

```
ExecutorService executor = Executors.newFixedThreadPool(4);

// submitting tasks

executor.shutdown();

boolean terminated = executor.awaitTermination(60,
TimeUnit.MILLISECONDS);

if (terminated) {
   System.out.println("The executor was successfully stopped");
} else {
   System.out.println("Timeout elapsed before termination");
}
```

An example: names of threads and tasks

we create one executor with a pool consisting of four threads. We submit ten tasks to it and then analyze the results. Each task prints the name of a thread that executes it, as well as the name of the task.

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

public class ExecutorDemo {
   private final static int POOL_SIZE = 4;
   private final static int NUMBER_OF_TASKS = 10;

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

      for (int i = 0; i < NUMBER_OF_TASKS; i++) {
         int taskNumber = i;
         executor.submit(() -> {
            String taskName = "task-" + taskNumber;
```

```
            String threadName = Thread.currentThread().getName();
            System.out.printf("%s executes %s\n", threadName, taskName);
        });
    }

    executor.shutdown();
  }
}
```

If you launch this program many times, you will get a different output. Below is one of the possible outputs:

```
pool-1-thread-1 executes task-0
pool-1-thread-2 executes task-1
pool-1-thread-4 executes task-3
pool-1-thread-3 executes task-2
pool-1-thread-3 executes task-7
pool-1-thread-3 executes task-8
pool-1-thread-3 executes task-9
pool-1-thread-1 executes task-6
pool-1-thread-4 executes task-5
pool-1-thread-2 executes task-4
```

It clearly demonstrates the executor uses all four threads to solve the tasks. The number of solved tasks by each thread can vary. There are no guarantees what we'll get.

If you do not know how many threads are needed in your pool, you can take the number of available processors as the pool size.

```
int poolSize = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(poolSize);
```

——————————————————— Runtime class
———————————————————

In Java, the Runtime class is a singleton class from the java.lang package that provides access to the Java runtime system. It allows you to interact with the Java Virtual Machine (JVM) environment and provides methods to manage the application's execution environment. Some common uses of the Runtime class include:

- **Getting Runtime Information**: You can obtain information about the runtime environment, such as available processors, total memory, free memory, and maximum memory.
- **Executing External Processes**: You can start new processes, execute commands in the operating system's command shell, and manage the input and output streams of these processes.
- **Halting and Exiting the JVM**: You can halt the JVM or

terminate the current Java application programmatically.

- **Managing Shutdown Hooks**: You can register shutdown hooks that are executed when the JVM is shutting down.

Here's a basic example demonstrating some common uses of the Runtime class:

```
public class Main {
    public static void main(String[] args) throws Exception {
        // Getting Runtime Information
        Runtime runtime = Runtime.getRuntime();
        System.out.println("Available Processors: " +
runtime.availableProcessors());
        System.out.println("Total Memory: " + runtime.totalMemory());
        System.out.println("Free Memory: " + runtime.freeMemory());
        System.out.println("Max Memory: " + runtime.maxMemory());

        // Executing External Processes
        Process process = runtime.exec("echo Hello, World!");
        process.waitFor();
        System.out.println("Exit Value: " + process.exitValue());

        // Halting and Exiting the JVM
        runtime.exit(0);
    }
}
```

```
Available Processors: 8
Total Memory: 2058354688
Free Memory: 189581024
Max Memory: 3817865216
Hello, World!
Exit Value: 0
```

———————————————————————————————————

Types of executors

We have considered the most used executor with the fixed size of the pool. Here are a few more types:

- **An executor with a single thread**

The simplest executor has only a single thread in the pool. It may be enough for async execution of rarely submitted and small tasks.
ExecutorService executor = Executors.newSingleThreadExecutor();

Important: one thread may not have time to process all incoming tasks, and the queue will grow significantly, consuming all the memory.

- **An executor with a growing pool**

There is also an executor that automatically increases the number of threads as needed and reuse previously constructed threads.
ExecutorService executor = Executors.newCachedThreadPool();

It can typically improve the performance of programs that perform many short-lived asynchronous tasks. But it can also lead to problems when the number of threads increases too much. It is preferable to choose the fixed thread-pool executor whenever possible.

- **An executor that schedules a task**

If you need to perform the same task periodically or only once after the given delay, use the following executor:

ScheduledExecutorService executor =
Executors.newSingleThreadScheduledExecutor();

The method scheduleAtFixedRate submits a periodic Runnable task that becomes enabled first after the given initDelay, and subsequently with the given period.

Here is a quick example with scheduling:

```
ScheduledExecutorService executor =
Executors.newSingleThreadScheduledExecutor();
executor.scheduleAtFixedRate(() ->
      System.out.println(LocalTime.now() + ": Hello!"), 1000, 1000,
TimeUnit.MILLISECONDS);
```

Here is a fragment of the output:

```
02:30:06.375392: Hello!
02:30:07.375356: Hello!
02:30:08.375376: Hello!
...and even more...
```

It can be stopped as we did before.
This kind of executor also has a method named schedule that starts a task only once after the given delay and another
method scheduleWithFixedDelay that starts the task with a fixed wait after the previous one is completed.

Exception handling

In our examples, we often ignore error handling to simplify code.

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

```
executor.submit(() -> System.out.println(2 / 0));
```

It does not print anything at all, including the exception! This is why it is common practice to wrap a task in the try-catch block not to lose the exception.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    try {
        System.out.println(2 / 0);
    } catch (Exception e) {
        e.printStackTrace();
    }
});
```

Now you will see the exception. In real applications, it is better to use some kind of logging to output it. Note that the executor will still work after the exception because it dynamically creates a new thread.