Introdução

Nesse módulo aprendemos como podemos deixar nossas aplicações mais performáticas ao implementar técnicas como **concorrência** e **paralelismo**.

Multi thread

Threads permitem que múltiplas funções sejam executadas no sistema operacional. Quando usamos o Java, automaticamente usamos threads, pois o Java foi pensado para executar tarefas em paralelo.

Threads se tornam extremamente importantes quando temos uma aplicação sendo usada por mais de uma pessoa, como é o caso das aplicações web. Isso acontece porque o servidor deve ser capaz de responder várias requisições ao mesmo tempo e se isso não acontecesse, os usuários poderiam ter uma péssima experiência com a lentidão nas respostas das requisições

Quando executamos qualquer programa Java, a **JVM** cria automaticamente a primeira thread para permitir a execução do programa. No entanto, podemos criar mais threads para adicionar paralelismo na aplicação:

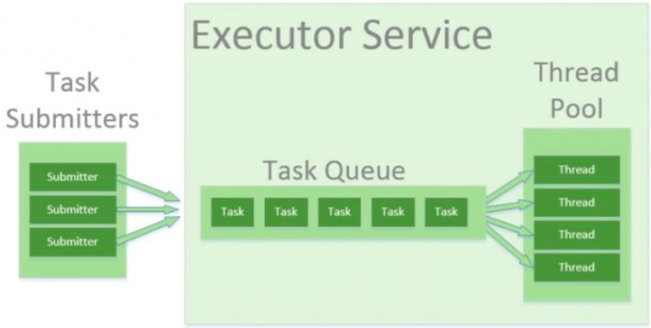
```
public class Main {
    public static void main(String[] args) {
        List<Integer> numeros = List.of(1,2,3,4,5,6,7,8,9,10);

        createThread("Thread 1", () -> {
            numeros.forEach(numero -> {
                threadSleep(numero * 100);
                System.out.println(Thread.currentThread().getName() + "
- Numero: " + numero * 2);
            });
        });

        createThread("Thread 2", () -> {
            numeros.forEach(numero -> {
                threadSleep(numero * 100);
                System.out.println(Thread.currentThread().getName() + "
- Numero: " + numero);
            });
        });
    }

    private static void threadSleep(long milliseconds) {
        try {
            Thread.sleep(milliseconds);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    private static void createThread(String threadName, Runnable
runnable) {
        final Thread thread = new Thread(runnable);
        thread.setName(threadName);
        thread.start();
    }
}
```

No exemplo anterior, nós criamos duas threads que serão executadas paralelamente, sendo impossível garantir que a saída de cada print será sempre a mesma.

Thread Pool

Thread Pool (**piscina de threads**) é um padrão que ajuda a otimizar o uso de recursos em aplicações multi thread e ainda determina limites para o paralelismo da aplicação. Esse padrão é necessário, pois a criação de Threads em tempo de execução é custoso ao processador e pode levar ao efeito contrário que queremos, causando lentidão ao sistema e possíveis encerramentos inesperados da aplicação.

Quando usamos thread pool, a ideia é separar as execuções paralelas e dividi-las em tarefas que serão submetidas a um grupo de threads previamente criadas, permitindo o reuso delas.



Na imagem anterior, temos um exemplo de tarefas sendo submetidas a um thread pool que aloca 4 threads para a aplicação.

No Java temos as interfaces "**Executor**" e "**ExecutorService**" que são usadas para trabalhar com diferentes implementações de thread pool. A interface "**Executor**" especifica o método "**execute**" que submete uma instância de "Runnable" para a execução.

```
Executor executor = Executors.newFixedThreadPool(2);

executor.execute(() -> System.out.println("Thread em execução: "
+ Thread.currentThread().getName()));
executor.execute(() -> System.out.println("Thread em execução: "
+ Thread.currentThread().getName()));
executor.execute(() -> System.out.println("Thread em execução: "
+ Thread.currentThread().getName()));
```

No exemplo acima criamos um pool de 2 threads e o resultado será que cada print pode ser executado em uma das duas:

```
Thread em execução: pool-1-thread-1
Thread em execução: pool-1-thread-2
Thread em execução: pool-1-thread-1
```

Future

A interface **Future** representa o resultado de um processamento assíncrono e provê métodos para checar se o processamento está completo, aguardar o processamento e recuperar o resultado. O método "submit" da interface "ExecutorService" retorna uma instância de "Future".

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

Future<String> future = executorService.submit(() -> "Thread: " +
Thread.currentThread().getName());
String resultado = future.get();
System.out.println(resultado);

executorService.shutdown();
```

No exemplo anterior foi criado um instância de "**ExecutorService**". O método "**submit**" retornará uma instância de Future de String. Note que "**ExecutorService**" é uma interface feita em cima de "**Executor**", pois estende dela e específica alguns outros métodos.

Para entender mais sobre a interface, recomendamos a documentação oficial dela: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html

Fork/Join

O framework de **fork/join** está presente desde a versão 7 do Java, onde são oferecidas ferramentas que auxiliam no aumento de desempenho no processamento paralelo, utilizando todas os núcleos do processador, tudo isso utilizando uma abordagem de divisão e conquista. Para entender mais detalhadamente sobre o framework, recomendamos esse tutorial oficial: https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

Resumidamente, o seguintes passos são executados: Uma tarefa maior é "quebrada" em sub-tarefas até que seja pequena o suficiente para rodar de forma assíncrona.

Na medida que as sub-tarefas são concluídas, seus resultados serão computados.

É utilizado um pool de threads chamado "**ForkJoinPool**" que gerencia as threads chamadas de "**ForkJoinWorkerThread**"