04

Distribuindo comandos e tratamento de erro

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/threads2/capitulo-5.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Distribuindo comandos

Vamos continuar com o nosso projeto **servidor-tarefas** e, claro, aprender novos recursos sobre threads. Criamos a classe `DistribuirTarefas` com o objetivo de receber os comandos do cliente. Isso já está acontecendo, implementamos a funcionalidade através do `switch`, que não é tão elegante mas resolve o nosso problema:

```
switch (comando) {
    case "c1": {
        saidaCliente.println("Confirmação do comando c1");
        break;
    }
    case "c2": {
        saidaCliente.println("Confirmação do comando c2");
        break;
    }
    case "fim": {
        saidaCliente.println("Desligando o servidor");
        servidor.parar();
        return; //saindo do metodo
    }
    default: {
        saidaCliente.println("Comando não encontrado");
    }
}
```

O que não está acontecendo é a distribuição em si! A ideia dessa classe é que cada comando será executado em uma nova thread, pois o comando pode executar algo demorado, como um cálculo pesado, integrações com outros sistema ou acesso ao banco de dados. Usando uma nova thread, não vamos bloquear o recebimento de novos comandos através do mesmo cliente. Bora implementar?

Cada comando, um Runnable

Vamos simular os dois comandos que fazem parte do nosso *protocolo*, `c1` e `c2`. Cada comando será a tarefa de uma thread. Aqui não há novidade, usamos a nossa velha conhecida interface `Runnable`:

```
public class ComandoC1 implements Runnable {

    @Override
    public void run() {
        // faz algo bem demorado
    }
}
```

E o `ComandoC2` :

```
public class ComandoC2 implements Runnable {

@Override
    public void run() {
        // faz algo bem demorado
    }
}
```

Cada comando deve devolver o resultado da execução, ou pelo menos informar ao cliente que a execução foi finalizada. Para tal, recebemos no construtor a saída do cliente:

```
public class ComandoC1 implements Runnable {

    private PrintStream saida;

    public ComandoC1(PrintStream saida) {
        this.saida = saida;
    }

    @Override
    public void run() {

        System.out.println("Executando comando c1");

        try {
            // faz algo bem demorado
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        //devolvendo resposta para o cliente
        saida.println("Comando c1 executado com sucesso!");
    }
}
```

Repare que usamos o `Thread.sleep(20000)` para simular a demora na execução. O `ComandoC2` ficará igual a essa, mudando somente as impressões, mas mais para frente faremos alterações nele. Por enquanto, os dois comandos são iguais.

Executando as threads

Com os comandos implementados, podemos pensar na criação das threads. Aqui também não há novidade e poderíamos criar uma nova para cada comando enviado pelo cliente. Já sabemos que devemos ter cuidado na hora de criar novas threads, pois há um custo operacional envolvido. Resolvemos essa questão através do pool que já temos criado, que também utilizaremos agora.

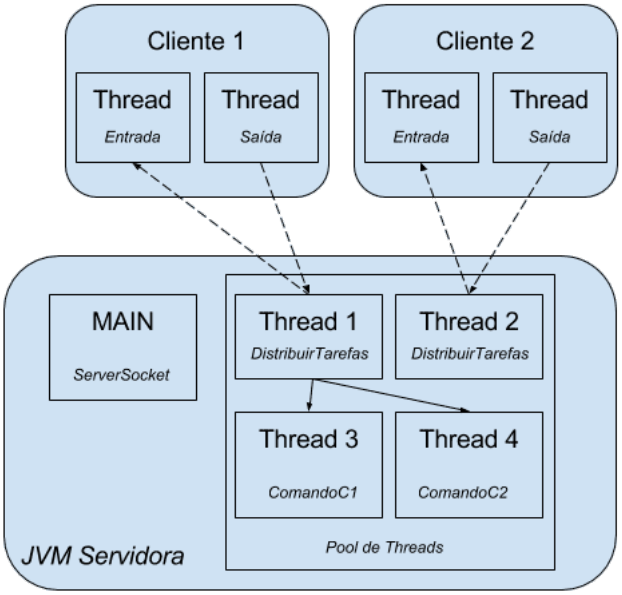
Quando criarmos a classe `DistribuirTarefas` , passaremos o pool (`threadPool`) como primeiro argumento do construtor:

```
// na classe ServidorTarefas
// no método rodar()
DistribuirTarefas distribuidor = new DistribuirTarefas(threadPool, socket, this);
```

Na classe `DistribuirTarefas`, podemos então utilizar o pool para rodar os nossos comandos:

```
public class DistribuirTarefas implements Runnable {

    private Socket socket;
    private ServidorTarefas servidor;
    private ExecutorService threadPool;

    public DistribuirTarefas(ExecutorService threadPool, Socket socket, ServidorTarefas servidor
        this.threadPool = threadPool; // recebendo o threadPool e associando a uma variável loca
        this.socket = socket;
        this.servidor = servidor;
    }

    @Override
    public void run() {

        // restante do código omitido

        switch (comando) {
            case "c1": {
                saidaCliente.println("Confirmação do comando c1");
                ComandoC1 c1 = new ComandoC1(saidaCliente);
                this.threadPool.execute(c1);
                break;
            }
            case "c2": {
                saidaCliente.println("Confirmação do comando c2");
                ComandoC2 c2 = new ComandoC2(saidaCliente);
                this.threadPool.execute(c2);
                break;
            }
            // outro case e default omitidos
        }
        // restante do código omitido
    }
}
```

Isso já é suficiente para distribuir os comandos em novas threads. Para deixar mais claro, segue um desenho que mostra a nossa arquitetura no lado do servidor:



Que tal testar exatamente esse desenho? Temos dois clientes conectados e o cliente 1 enviou dois comandos, `c1` e `c2`.

Para ver a nossa distribuição funcionar, vamos utilizar `FixedThreadPool` com um máximo de 4 threads. Para isso, alteraremos o construtor da classe `ServidorTarefas`:

```
public ServidorTarefas() throws IOException {
    System.out.println("---- Iniciando Servidor ----");
    this.servidor = new ServerSocket(12345);
    //usando FixedThreadPool, max 4 Threads
    this.threadPool = Executors.newFixedThreadPool(4);
    this.estaRodando = new AtomicBoolean(true);
}
```

Isso significa que nosso pool não vai permitir mais do que 4 threads. Com dois clientes conectados, já usamos duas threads, basta então enviar dois comandos para esgotar o pool. Qualquer outro comando enviado não será confirmado até uma thread ser liberada.

E se um erro acontece na threads?

Criamos os comandos com a motivação de fazer um processamento demorado, por exemplo, um cálculo pesado ou a integração com outros sistemas. Com certeza, em algum momento, podem acontecer exceções nesse processamento. A pergunta é, como vamos lidar com as exceções que vão acontecer em nossas threads?

Antes de implementar um tratamento de erro no nosso projeto **servidor-tarefas**, vamos executar um teste separado. Podemos aproveitar o projeto **experimento**, que usamos para entender a palavra chave `volatile`. Nele, criamos uma thread separada e agora geraremos uma exceção explicitamente nela:

```
// projeto experimento

public class ServidorDeTeste {

// atributo e main comentados

    private void rodar() {
```

```
        try {
            new Thread(new Runnable() {

                public void run() {
                    System.out.println("Servidor começando, estaRodando = "
                            + estaRodando);
                    while (!estaRodando) {
                    }

                    if (estaRodando) {
                        throw new RuntimeException("Deu erro na thread...");
                    }

                    System.out.println("Servidor rodando, estaRodando = "
                            + estaRodando);

                    while (estaRodando) {
                    }

                    System.out.println("Servidor terminando, estaRodando = "
                            + estaRodando);
                }
            }).start();
        } catch (Exception e) {
            System.out.println("Catch na thread MAIN " + e.getMessage());
        }
    }

    // método alterandoAtributo comentado
}
```

Repare que estamos jogando uma `RuntimeException` dentro do método `run`. Além disso, toda a thread está envolvida em um bloco `try-catch`. Em outras palavras, não só jogamos uma exceção, como também tentamos resolvê-la.

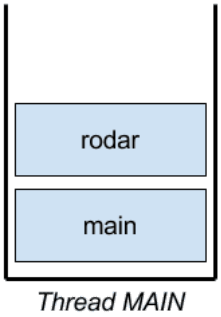
Vamos testar o código e rodar a classe `ServidorDeTeste`. O esperado seria que o bloco `catch` capture a exceção e imprima a mensagem no console, mas veja a saída:

```
Servidor começando, estaRodando = false
Main alterando estaRodando = true
Exception in thread "Thread-0" java.lang.RuntimeException: deu errado ....
    at br.com.alura.threads.ServidorDeTeste$1.run(ServidorDeTeste.java:23)
    at java.lang.Thread.run(Thread.java:745)
Main alterando estaRodando=false
```

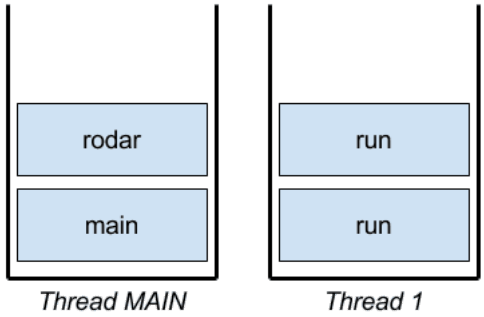
Repare que a mensagem da exceção não apareceu! A exceção *explodiu* no console e a thread `main` parece não estar ligando muito para isso!

Cada thread a sua pilha

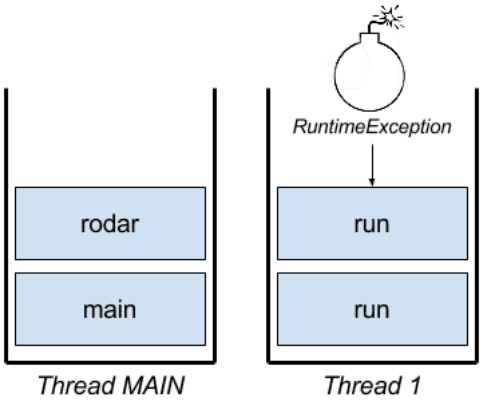
Quando inicializamos o nosso programa, a JVM cria automaticamente a thread `MAIN` e ela sempre começa no método `main` (*que surpresa*). O `main` é o primeiro método na pilha de métodos! Se chamamos um novo método a partir do `main`, este fica no topo dessa pilha e assim por diante:



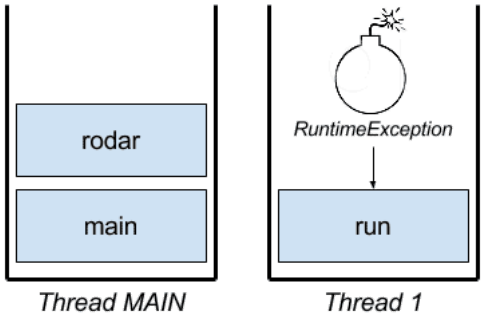
Como inicializamos uma nova thread, essa também terá a sua pilha, independente da thread `MAIN`. A diferença é que essa pilha começa com o método `run`, da classe Thread, que por sua vez chama o método `run` da tarefa (`Runnable`):



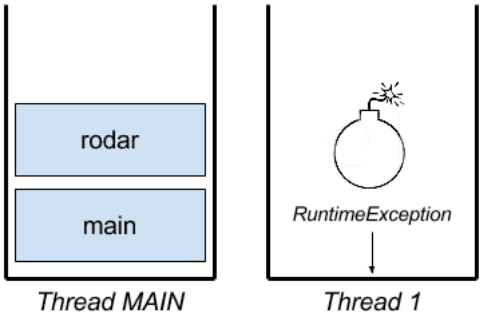
Essa pilha também é visível na saída do console que recebemos antes, pois mostra o caminho que a exceção traçou. Quando ocorre uma exceção no nosso código, ele se comporta como um bomba que cai em cima da pilha.



Se não há nenhum bloco `try-catch`, essa bomba vai remover o método da pilha:



E se não existe mais nenhum método na pilha, o rastro da exceção aparece no console:




```
Exception in thread "Thread-0" java.lang.RuntimeException: deu errado ....
        at br.com.alura.threads.ServidorDeTeste$1.run(....:23)
        at java.lang.Thread.run(Thread.java:745)
```

Repare que a exceção anda totalmente independente da outra pilha. Mesmo com a exceção na `Thread 1`, a thread `MAIN` continua rodando. Também não adianta tentar pegar a exceção através de um `try-catch`!

Será que não há uma outra forma de centralizar o tratamento de erro ou é realmente preciso espalhá-lo em cada tarefa/comando?

Capturando exceções

A resposta está na classe `Thread`, no método `setUncaughtExceptionHandler`. Podemos passar um objeto que recebe a exceção, caso aconteça, para a thread. Achamos o lugar central para o tratamento!

Vamos testar o `ExceptionHandler`, ainda no projeto **experimento**:

```
// na classe ServidorDeTeste, do projeto experimento

private void rodar() {
    // sem try - catch

    Thread thread = new Thread(new Runnable() {

        public void run() {
            // código que gera uma exceção omitido
        }
    });

    // passando o objeto com a responsabilidade de tratamento de erro
    thread.setUncaughtExceptionHandler(new TratadorDeExcecao());

    thread.start();
}
```

Repare que apagamos o `try-catch` e usamos uma variável para a thread.

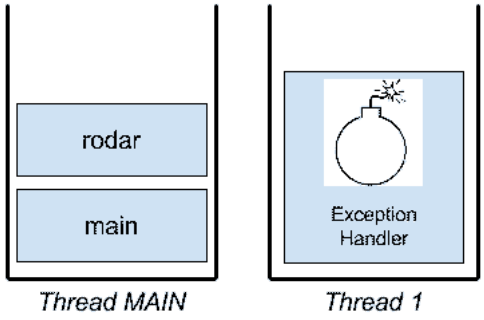
A classe `TratadorDeExcecao` deve implementar a interface `UncaughtExceptionHandler`, que possui um único método:

```
public class TratadorDeExcecao implements UncaughtExceptionHandler {

    @Override
    public void uncaughtException(Thread t, Throwable e) {
```

```
            System.out.println("Deu exceção na thread " + t.getName() + ", "
                    + e.getMessage());
        }
    }
```

Já podemos testar o código e rodar novamente a classe `ServidorDeTeste`. A saída no console mostrará a mensagem de erro, mas agora sem traço da pilha! A exceção foi repassada para a classe `TratadorDeExcecao`:

```
Servidor começando, estaRodando = false
Main alterando estaRodando = true
Deu exceção na thread Thread-0, Deu erro na thread...
Main alterando estaRodando = false
```



Ótimo, mas será que vai ajudar no nosso projeto **servidor-tarefas**?

Usando a fábrica de threads

Agora está na hora de aplicar essa forma de tratamento de exceções em threads no nosso projeto **servidor-tarefas**, mas ainda temos um problema para resolver. No exemplo anterior, criamos uma instância da classe `Thread` na mão, no entanto no projeto **servidor-tarefas** usamos um pool de threads! Não temos acesso direto às instâncias de threads.

O problema é, sem ter esse acesso, como vamos passar o `ExceptionHandler`? A solução está no método que cria o pool de threads. Cada método da classe Executors -- tanto faz se criarmos um pool fixo ou *cached* -- é sobrecarregado e pode receber um segundo parâmetro:

```
//construtor da classe ServidorTarefas

public ServidorTarefas() throws IOException {
    System.out.println("---- Iniciando Servidor ----");
    this.servidor = new ServerSocket(12345);
    this.threadPool = Executors.newFixedThreadPool(4, new FabricaDeThreads());
    this.estaRodando = new AtomicBoolean(true);
}
```

A fábrica deve implementar a interface `ThreadFactory` e possuir um método que será chamado através do pool, se ele precisar de uma nova thread. Nesse método, vamos criar a thread e adicionar o *tratador de exceções*. Aproveitamos também para dar um nome à thread:

```
public class FabricaDeThreads implements ThreadFactory {
```

```
    private static int numero = 1;

    @Override
    public Thread newThread(Runnable r) {

        Thread thread = new Thread(r, "Thread Servidor Tarefas " + numero);

        numero++;

        thread.setUncaughtExceptionHandler(new TratadorDeExcecao());

        return thread;
    }
}
```

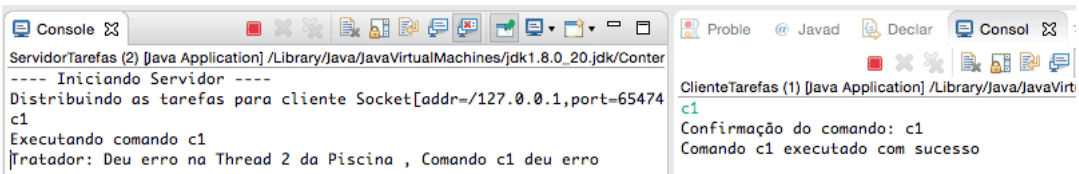
Está quase tudo pronto para testar, mas não podemos nos esquecer de copiar a classe `TratadorDeExcecao` do projeto **experimento** para o projeto **servidor-tarefas**.

Para realmente ver o tratador funcionando, forçaremos uma exceção na classe `ComandoC2`:

```
public class ComandoC2 implements Runnable {

    // atributo e construtor omitido

    @Override
    public void run() {

        // toda implementação omitida

        throw new RuntimeException("Exception no comando c2");
    }
 }
```

Ao rodar o nosso servidor e conectar um cliente, devemos enviar o comando `c2`. No console do servidor deve aparecer a saída do `ExceptionHandler`:

```
---- Iniciando Servidor ----
Distribuindo as tarefas para o cliente Socket[addr=/127.0.0.1,port=64388,localport=12345]
Comando recebido c2
c2
Executando comando c2
Deu exceção na thread Thread Servidor Tarefas 2, Exception no comando c2
```



O que aprendemos?

- Cada thread possui a sua pilha de métodos.

- O tratamento de exceções deve ser específico para cada pilha.
- Podemos plugar um `UncaughtExceptionHandler` para centralizar o tratamento.
- O pool de threads oferece uma fábrica de threads para personalizar a criação da thread.