03

Melhorando o cliente

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/threads2/capitulo-3.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

O nosso cliente até agora é bem simples e envia um comando através do `OutputStream` do `Socket` para o servidor.

```
public class ClienteTarefas {

    public static void main(String[] args) throws Exception {

        Socket socket = new Socket("localhost", 12345);
        System.out.println("conexão estabelecida");

        //enviando comando c1 para servidor
        PrintStream saida = new PrintStream(socket.getOutputStream());
        saida.println("c1");

        //aguardando enter
        Scanner teclado = new Scanner(System.in);
        teclado.nextLine();

        //fechando recursos
        saida.close();
        teclado.close();
        socket.close();
    }
}
```

O nosso objetivo agora é enviar qualquer comando pelo cliente e também receber uma resposta do servidor com uma confirmação do comando. Repare que o cliente precisa executar duas tarefas ao mesmo tempo! Isso te lembra de alguma coisa? Exato, vamos utilizar as thread para tal.

Capturando a entrada

Para capturar a entrada, vamos utilizar o nosso teclado que já temos em mãos. Ou seja, através do `Scanner` , vamos ler os comandos do teclado e essa entrada enviaremos para o servidor:

```
public class ClienteTarefas {

    public static void main(String[] args) throws Exception {

        Socket socket = new Socket("localhost", 12345);
        System.out.println("Conexão Estabelecida");
        PrintStream saida = new PrintStream(socket.getOutputStream());

        Scanner teclado = new Scanner(System.in);
        while (teclado.hasNextLine()) {
            String linha = teclado.nextLine();
            saida.println(linha);
```

```
        }

        saida.close();
        teclado.close();
        socket.close();
    }
}
```

Também já vamos implementar uma condição de saída. Quando digitamos apenas `ENTER`, sairemos do laço:

```
while (teclado.hasNextLine()) {
    String linha = teclado.nextLine();

    if (linha.trim().equals("")) {
        break;
    }

    saida.println(linha);
}
```

Lendo dados do Servidor

O nosso servidor também pode devolver dados para o cliente, por exemplo, a confirmação do comando ou algum resultado de um comando submetido. Para podermos receber os dados, devemos utilizar o `InputStream` do nosso cliente:

```
public class ClienteTarefas {

    public static void main(String[] args) throws Exception {

        // enviando comandos
        System.out.println("Recebendo dados do servidor");
        Scanner respostaServidor = new Scanner(socket.getInputStream();

        while (respostaServidor.hasNextLine()) {
            String linha = respostaServidor.nextLine();
            System.out.println(linha);
        }

        respostaServidor.close();
        // fechando outros recursos
    }
}
```

Juntando e executando todo o código, temos um problema porque o nosso envio de comandos bloqueia o recebimento de dados do servidor. Em outras palavras, nunca poderemos receber uma resposta do servidor enquanto estivermos enviando dados. É exatamente essa a nossa motivação para utilizar threads também no lado do cliente.

Usando threads no cliente

Agora não há muitas novidades e já vimos isso em outros exemplos. Vamos separar o recebimento e o envio dos dados, cada um em uma thread. Mãos à obra!

Poderíamos criar classes separadas para cada tarefa (`Runnable`), mas vamos usar um atalho, que é criar duas classes anônimas. Repare no código abaixo que já instanciamos o `Runnable` na criação da thread:

```
Thread threadEnviaComando = new Thread(new Runnable() {

    @Override
    public void run() {

        try {
            System.out.println("Pode enviar comandos!");
            PrintStream saida = new PrintStream(socket.getOutputStream());

            Scanner teclado = new Scanner(System.in);
            while (teclado.hasNextLine()) {

                String linha = teclado.nextLine();

                if (linha.trim().equals("")) {
                    break;
                }

                saida.println(linha);
            }

            saida.close();
            teclado.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
});
```

E a definição da thread para receber a resposta do servidor:

```
Thread threadRecebeResposta = new Thread(new Runnable() {

    @Override
    public void run() {

        try {
            System.out.println("Recebendo dados do servidor");
            Scanner respostaServidor = new Scanner(socket.getInputStream());

            while (respostaServidor.hasNextLine()) {
                String linha = respostaServidor.nextLine();
                System.out.println(linha);
            }

            respostaServidor.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
```

```
        }
    });
```

Ao final de todo o código dessas duas threads, podemos inicializá-las:

```
threadRecebeResposta.start();
threadEnviaComando.start();
socket.close();
```

Juntando as Threads

Com o servidor rondando, podemos executar o cliente. No entanto recebemos uma exceção:

```
java.net.SocketException: Socket is closed
```

Aliás, recebemos duas vezes essa exceção, uma para cada thread. O que está acontecendo?

O problema é o seguinte: estamos inicializando cada thread corretamente, mas o `Socket` é fechado na thread principal (`main`). Quando estamos começando a enviar e receber dados, é provável que a thread `main` já tenha fechado o `Socket` !

Devemos parar a thread `main` e poderíamos utilizar novamente o método `Thread.sleep(..)` . Mas qual seria o tempo adequado para esperar? Além disso, já temos um critério de interrupção. Na hora de fazer a leitura, quando apertamos apenas `ENTER` , aí sim devemos parar.

Para resolver o nosso problema, podemos indicar à thread `main` esperar a execução enquanto a thread de leitura está rodando. Isso é feito através do método `join` . Veja o código:

```
// criação das threads omitida
threadRecebeResposta.start();
threadEnviaComandos.start();

//thread main vai esperar
threadEnviaComandos.join();

System.out.println("Fechando o socket do cliente");
socket.close();
```

Quando a thread `main` executa o método `join` , ela sabe que precisa esperar a execução da thread que envia os comandos . A thread `main` ficará esperando até a outra thread acabar.

Quando rodarmos o nosso cliente, não deverá aparecer o `System.out.println("Fechando o socket do cliente")` . Apenas quando finalizarmos o envio dos comandos com um simples `ENTER` , terminaremos a thread de enviar comandos e assim a thread `main` poderá continuar.

Segue uma vez o cliente completo:

```
public class ClienteTarefas {

    public static void main(String[] args) throws Exception {
        Socket socket = new Socket("localhost", 12345);
        System.out.println("Conexão Estabelecida");

        Thread threadEnviaComando = new Thread(new Runnable() {

            @Override
            public void run() {

                try {
                    System.out.println("Pode enviar comandos!");
                    PrintStream saida = new PrintStream(socket.getOutputStream());

                    Scanner teclado = new Scanner(System.in);
                    while (teclado.hasNextLine()) {

                        String linha = teclado.nextLine();

                        if (linha.trim().equals("")) {
                            break;
                        }

                        saida.println(linha);
                    }

                    saida.close();
                    teclado.close();

                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
        });

        Thread threadRecebeResposta = new Thread(new Runnable() {

            @Override
            public void run() {

                try {
                    System.out.println("Recebendo dados do servidor");
                    Scanner respostaServidor = new Scanner(socket.getInputStream());

                    while (respostaServidor.hasNextLine()) {
                        String linha = respostaServidor.nextLine();
                        System.out.println(linha);
                    }

                    respostaServidor.close();

                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
        });
```

```
        threadRecebeResposta.start();
        threadEnviaComando.start();

        threadEnviaComando.join();

        System.out.println("Fechando o socket do cliente");

        socket.close();
    }
}
```

Confirmando o recebimento de comandos

Para realmente testar o recebimento dos dados, também precisamos ajustar o servidor. Em detalhes, é necessário mexer na classe `DistribuirTarefas`. Nela vamos devolver uma confirmação para o cliente. Para tal, pegaremos o `OutputStream`:

```
OutputStream outputCliente = socket.getOutputStream();
PrintStream saida = new PrintStream(outputCliente);
```

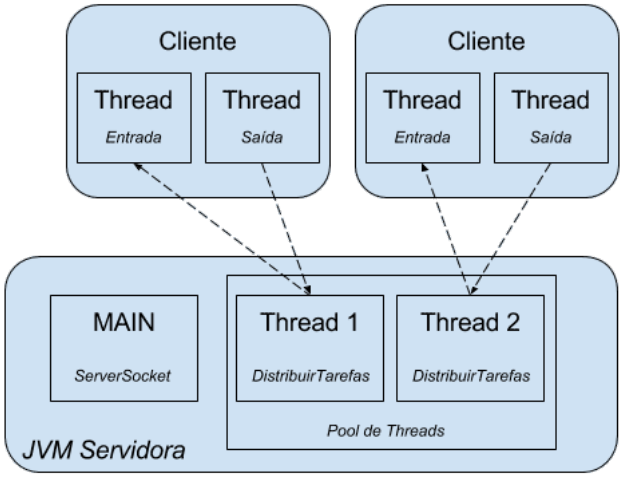
Com a saída em mãos, podemos confirmar o comando recebido dentro do laço `while`. Vamos testar o comando através de `switch`:

```
//classe DistribuirTarefas

@Override
public void run() {

    try {
        System.out.println("Distribuindo as tarefas para o cliente " + socket);

        Scanner entradaCliente = new Scanner(socket.getInputStream());
        PrintStream saidaCliente = new PrintStream(socket.getOutputStream());

        while (entradaCliente.hasNextLine()) {

            String comando = entradaCliente.nextLine();
            System.out.println("Comando recebido " + comando);

            switch (comando) {
                case "c1": {
                    // confirmação do o cliente
                    saidaCliente.println("Confirmação do comando c1");
                    break;
                }
                case "c2": {
                    saidaCliente.println("Confirmação do comando c2");
                    break;
                }
                default: {
                    saidaCliente.println("Comando não encontrado");
                }
```

```
            }

            System.out.println(comando);
        }

        saidaCliente.close();
        entradaCliente.close();

    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Repare que já podemos receber dois comandos `c1` e `c2` (os nomes foram inventados, poderia ser qualquer outra sigla!). Ambos os comandos são confirmados através de uma mensagem de resposta.

Nesse capítulo melhoramos apenas o nosso cliente, como apresentado na imagem abaixo:



O que aprendemos?

- Usar o `Runnable` através de classes anônimas.
- O método `thread.join()` faz com que a thread que executa espere até o outro acabar.
- só faz sentido usar um pool de threads quando realmente queremos reaproveitar uma thread