ECM251 – Linguagens de Programação I

Aula 22 – L1/1, L2/1 e L3/1

Engenharia da Computação – 3ª série

Cliente-Servidor Java Sockets e Threads (L1/1, L2/1 e L3/1)

2024

Aula 22 – L1/1 e L2/1

Horário

Terça-feira: 2 x 2 aulas/semana

- L1/1 (07h40min-09h20min): *Prof. Calvetti*;
- L1/2 (09h30min-11h10min): *Prof. Calvetti*;
- L2/1 (07h40min-09h20min): *Prof. Evandro*;
- L2/2 (11h20min-13h00min): *Prof. Calvetti*;
- L3/1 (09h30min-11h10min): *Prof. Evandro*;
- L3/2 (11h20min-13h00min): *Prof. Evandro.*

Tópico

- Expressão *Lambda* em Java

Expressão Lambda em Java

Definição

- Uma **Expressão *Lambda*** em **Java** é uma característica introduzida no **Java 8** que permite criar funções anônimas de forma concisa;
- Facilitam a passagem de funcionalidades como argumentos para métodos ou a definição de métodos em interfaces funcionais, tornando o código mais limpo e legível.
- A sintaxe básica de uma **Expressão *Lambda*** é:

(*parâmetros*) -> *expressão*

- Onde: *parâmetros* são os parâmetros do método, sem a necessidade de declarar o tipo explicitamente; e *expressão* é o código executado quando o **método *lambda*** é chamado.

Expressão Lambda em Java

Exemplos

1. ***Lambda*** simples:

```
1 /* Usando uma expressão lambda para criar uma função
2    que adiciona dois números inteiros */
3 Operacao soma = (a, b) -> a + b;
4 int resultado = soma.executar(5, 3); // resultado é 8
5
```

2. ***Lambda*** com tipos explícitos:

```
1 // Você também pode especificar os tipos de parâmetros se desejar
2 Comparador<String> comparador = (String s1, String s2) -> s1.compareTo(s2);
3 // resultado é um valor que indica a ordem lexicográfica
4 int resultado = comparador.compare("maçã", "banana");
5
```

3. Filtragem com ***Lambda***:

```
1 List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
2 List<Integer> pares = numeros.stream().filter(n -> n % 2 == 0).collect(Collectors.toList());
3
```

Conclusões

- **Expressões *Lambda*** são frequentemente usadas em conjunto com *streams*, interfaces funcionais e métodos de alto nível, como ***forEach***, ***filter***, ***map***, ***reduce*** etc.;
- Tornam o código mais conciso e mais legível ao remover a necessidade de criar **classes anônimas** e definir métodos com parâmetros e retornos específicos para tarefas simples.

Tópico

- Funções Anônimas em Java

Funções Anônimas em Java

Definição

- Em **Java**, as **Funções Anônimas**, ou **Métodos Anônimos**, são frequentemente associadas a **Interfaces Funcionais** e são implementadas como **Classes Anônimas** que substituem métodos específicos da interface funcional;
- Isso é especialmente útil quando deseja-se passar um método como argumento para um outro método ou quando deseja-se criar *call-backs*;

Funções Anônimas em Java

Exemplos

4. **Método Anônimo** em **Java**, onde a interface funcional **Operacao** tem um método abstrato chamado *executar( )* e uma classe anônima instanciando uma interface, fornecendo uma implementação para o método *executar( )*:

```
1  // Definindo uma interface funcional com um único método abstrato
2  interface Operacao
3  {  int executar(int a, int b);
4  }
5
6  public class ExemploFuncaoAnonima
7  {  public static void main(String[] args)
8     {  // Criando uma instância da interface funcional usando uma classe anônima
9        Operacao adicao = new Operacao()
10       {  @Override
11          public int executar(int a, int b)
12          {  return a + b;
13          }
14       };
15
16       // Usando a função anônima
17       int resultado = adicao.executar(5, 3);
18       System.out.println("Resultado: " + resultado);
19    }
20 }
21
```

Funções Anônimas em Java

Exemplos

5. Com o **Java 8** e a introdução de **Expressões *Lambda***, pode-se simplificar ainda mais o código, tornando-o mais conciso, por exemplo, refatorando o código anterior com **Expressão *Lambda***:

```
1  // Definindo uma interface funcional com um único método abstrato
2  interface Operacao
3  {  int executar(int a, int b);
4  }
5
6  public class ExemploFuncaoAnonima
7  {  public static void main(String[] args)
8     {  // Usando uma expressão lambda para criar uma função anônima
9        Operacao adicao = (a, b) -> a + b;
10       // Usando a função anônima
11       int resultado = adicao.executar(5, 3); // resultado é 8
12       System.out.println("Resultado: " + resultado);
13    }
14 }
15
```

Tópico

- *Threads*

Definição

- Termo originário do Inglês, ***Thread***, em Português, significa fio, linha, trilha e caminho, dependendo do contexto;
- Na área da computação, ***Thread*** é um recurso, do Sistema Operacional – SO, que permite que uma aplicação execute várias tarefas simultaneamente, cada uma representando um caminho diferente de execução dentro dessa aplicação, daí seu nome;
- Se existirem várias CPUs, ou vários núcleos de uma CPU, cada ***Thread*** pode ser executada por um(a) deles(as), simultaneamente aos outros de fato, por um processamento diferente, determinando uma **Execução Paralela** de tarefas;

Definição

- Se existir uma única CPU, ou um único núcleo da CPU disponível, o SO encarrega-se de alternar e intercalar a execução das diversas ***Threads*** ao longo do tempo, o que, numa alta frequência (velocidade), dar-se-á a falsa impressão de uma execução paralela de tarefas, porém, estabelecendo na realidade, uma **Execução Concorrente** de tarefas;
- Essa alternância entre as aplicações é que permite, por exemplo, "baixar arquivos", "escutar música" e "jogar" ao chamado "mesmo tempo" e é uma característica do SO denominado **Multitarefa**, pois o número de aplicações geralmente é sempre maior que o número de CPUs ou núcleos de CPU disponíveis;

Definição

- Nos SOs atuais, essa alternância entre as aplicações é uma característica denominada **Multitarefa Preemptiva**, onde é o SO quem decide o tempo máximo que cada tarefa pode utilizar a CPU e controla a fila de processos de execução estabelecida;
- Têm características de **Multitarefa Preemptiva** os seguintes SOs:
  - ✓ *Microsoft Windows*;
  - ✓ *GNU/Linux*;
  - ✓ *Apple macOs*;
  - ✓ *Apple iOs*; e
  - ✓ *Google Android*.

Definição

- Esse tempo de uso da CPU destinado a cada tarefa intercalada, chamado de ***Quantum*** ou ***Time-Slice***, é definido pelo SO, podendo ser redefinido de acordo com a prioridade do processo ou ***Thread***;
- O ***Scheduler*** é o componente do SO que define esse tempo de alternância e a ordem de execução das ***Threads***;
- Todo processo tem uma ***Thread*** ao menos, criada automaticamente quando o processo é iniciado, denominada de ***Thread*** **Principal**, ou seja, se nenhuma outra ***Thread*** for adicionada, o código da aplicação será executado somente pela ***Thread*** **Principal**, utilizando uma CPU ou um núcleo de CPU;

Definição

- Na Plataforma **Java**, a Java Virtual Machine – **JVM** é quem cria a ***Thread* Principal**, solicitando-a ao SO quando for executar uma aplicação **Java**;
- Novas ***Threads*** podem ser adicionadas pelo desenvolvedor da aplicação, por bibliotecas ou recursos da linguagem de programação, normalmente para executar uma determinada tarefa "em paralelo" com as outras;
- ***Threads*** de um mesmo processo usam uma única memória compartilhada, o que facilita a programação, mas podendo causar conflitos/erros, quando várias ***Threads*** tentam consultar/modificar uma mesma variável ao mesmo tempo.

Conclusões

- O código associado a uma ***Thread*** é que de fato é executado pelo processador;
- Se nenhum ***Thread*** adicional for criado de alguma forma, todo o código é executado pelo ***Thread Principal***;
- O ***Thread Principal*** é que inicia a execução do processo/aplicação e o desenvolvedor pode criar ***Threads*** adicionais, quando necessárias;
- Cada ***Thread*** pode executar uma tarefa diferente, com diferentes níveis de complexidade computacional, ou dividir uma tarefa grande em subtarefas, para processamento paralelo, podendo cada uma ser executada em seu próprio tempo/complexidade.

Tópico

- Criando *Threads* em Java

Definição

- Para o programador Java, o uso de ***Threads*** é importante por várias razões, sendo as principais:

  1. Melhor Desempenho: permitem que um programa Java execute tarefas em paralelo, aproveitando o poder de processadores multicore, o que pode levar a um melhor desempenho e uma resposta mais rápida do programa;

  2. Responsividade: criar ***Threads*** separadas, para tarefas demoradas, ajuda a manter os aplicativos gráficos e as interfaces com os usuários responsivas, pois uma tarefa longa poderia bloquear ou truncar, tornando-as não responsivas;

Definição

- Para o programador Java, o uso de ***Threads*** é importante por várias razões, sendo as principais:

3. Concorrência: são uma maneira de lidar com a concorrência em um programa, acessando e modificando recursos compartilhados, como bancos de dados ou variáveis de estado, ajudando a evitar problemas de concorrência;

4. Tarefas Paralelas: em tarefas que podem ser realizadas paralelamente, como cálculos intensivos ou busca em grandes conjuntos de dados, pois permitem que sejam executadas simultaneamente, economizando tempo;

Definição

- Para o programador Java, o uso de ***Threads*** é importante por várias razões, sendo as principais:

5. Resposta a Eventos: Em aplicativos que respondem a eventos, como servidores *web* ou sistemas de mensagens, podem ajudar a lidar com várias solicitações ou eventos simultaneamente;

6. Escalabilidade: Em sistemas que precisam lidar com um grande número de solicitações simultâneas, como servidores, podem ser usada para criar uma arquitetura escalável que pode atender a várias solicitações concorrentes;

Definição

- Para o programador Java, o uso de ***Threads*** é importante por várias razões, sendo as principais:

  7. Divisão de Tarefas: podem ser usadas para dividir um problema complexo em tarefas menores e independentes, tornando o código mais modular e mais fácil de manter; e

  8. Temporização: são úteis para lidar com tarefas que requerem temporização precisa, como atualização de gráficos em jogos ou monitoramento de eventos em tempo real.

Criando Threads em Java

Exemplo 6

- Gerar 1 bilhão de números aleatórios com 1 *Thread*:

```
1  import java.util.Random;
2  public class SimpleThreadExample
3  {  // Geracao de um bilhao de numeros aleatorios...
4     private static final long TOTAL_NUMEROS = 1_000_000_000L;
5
6     public static void main(String args[])
7     {  final int threads = 1;  // criar apenas 1 outra Thread
8        System.out.println("Missao: Gerar um bilhao de numeros aleatorios!");
9        System.out.printf("- Criando %d Thread(s) para isso!\n", threads);
10       new SimpleThreadExample().run(); // Uma Thread por vez (linear)
11    }
12
13    public void run()
14    {  System.out.println("- Iniciando Thread Principal...");
15       // Le o tempo do sistema no inicio do processamento
16       final double startTime = System.currentTimeMillis();
17       // Sorteia um numero (randomico)
18       Random rand = new Random();
19       // Eleva a 10a potencia cada numero para o processamento mais complexo
20       for(int i = 0; i < TOTAL_NUMEROS; i++)
21       {  Math.pow(rand.nextDouble(), 10);
22       }
23       // Tempo de processamento em segundos
24       final double totalSecs = (System.currentTimeMillis()-startTime)/1000.0;
25       System.out.println("- Encerrando o processamento...");
26       System.out.printf("Missao cumprida em %.2f segundos!\n", totalSecs);
27    }
28 }
```

Criando Threads em Java

Exemplo 6

- Gerar 1 bilhão de números aleatórios com 1 *Thread*:

```
1  import java.util.Random;
2  public class SimpleThreadExample
3  {  // Geracao de um bilhao de numeros aleatorios...
4     private static final long TOTAL_NUMEROS = 1_000_000_000L;
5
6     public static void main(String args[])
7     {  final int threads = 1;  // criar apenas 1 outra Thread
8        System.out.println("Missao: Gerar um bilhao de numeros aleatorios!");
9        System.out.printf("- Criando %d Thread(s) para isso!\n", threads);
10       new SimpleThreadExample().run(); // Uma Thread por vez (linear)
11    }
12
13    public void run()
14    {  System.out.println("- Iniciando Thread Principal...");
15       // Le o tempo do sistema no inicio do processamento
16       final double startTime = System.currentTimeMillis();
17       // Sorteia um numero (randomico)
18       Random rand = new Random();
19       // Eleva a 10a potencia cada numero para o processamento mais complexo
20       for(int i = 0; i < TOTAL_NUMEROS; i++)
21       {  Math.pow(rand.nextDouble(), 10);
22       }
23       // Tempo de processamento em segundos
24       final double totalSecs = (System.currentTimeMillis()-startTime)/1000.0;
25       System.out.println("- Encerrando o processamento...");
26       System.out.printf("Missao cumprida em %.2f segundos!\n", totalSecs);
27    }
28 }
```

```
Missao: Gerar um bilhao de numeros aleatorios!
- Criando 1 Thread(s) para isso!
- Iniciando Thread Principal...
- Encerrando o processamento...
Missao cumprida em 35,75 segundos!
```

Criando Threads em Java

Exemplo 7

- Gerar 1 bilhão de números aleatórios com várias *Threads*:

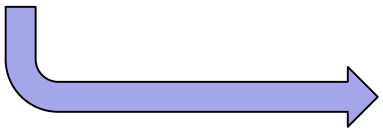
```
1  import java.util.Random;
2  public class MultipleThreadExample extends Thread
3  {  // Geracao de um bilhao de numeros aleatorios...
4     private static final long TOTAL_NUMEROS = 1_000_000_000L;
5     private final long numerosAGerar;
6
7     public MultipleThreadExample(long numerosAGerar)
8     {  this.numerosAGerar = numerosAGerar;
9        System.out.printf("----> Thread criada (%s)!\n", getName());
10    }
11
12    public static void main(String args[])
13    {  final int threads = args.length == 0 ? 1 : Integer.valueOf(args[0]);
14       final long numeroPorThread = (TOTAL_NUMEROS/threads);
15       System.out.printf("\nMissao: Gerar um bilhao de numeros aleatorios usando Threads!\n");
16       System.out.printf("-> Thread iniciada (%s)...\n", Thread.currentThread().getName());
17       System.out.printf("-> Criando +%d Thread(s) para isso:\n", threads);
18       // Le o tempo do sistema no inicio do processamento
19       final double startTime = System.currentTimeMillis();
20       for(int i = 0; i < threads; i++)
21       {  new MultipleThreadExample(numeroPorThread).start();   // Threads em paralelo!
22       }
23       // Tempo de processamento em segundos
24       final double totalSecs = (System.currentTimeMillis()-startTime)/1000.0;
25       System.out.printf("-> Thread finalizada (%s) em %.2fs!\n", Thread.currentThread().getName(), totalSecs);
26    }
27
```

Criando Threads em Java

Exemplo 7

- Gerar 1 bilhão de números aleatórios com várias *Threads*:

```
28    @Override
29    public void run()
30    {  System.out.printf("----> Iniciando Thread (%s)...\n", Thread.currentThread().getName());
31       final double startTime = System.currentTimeMillis();
32       // Sorteia um numero (randomico)
33       Random rand = new Random();
34       // Eleva a 10a potencia cada numero para o processamento mais complexo
35       for(int i = 0; i < numerosAGerar; i++)
36       {  Math.pow(rand.nextDouble(), 10);
37       }
38       final double totalSecs = (System.currentTimeMillis()-startTime)/1000.0;
39       System.out.printf("----> Encerrando Thread (%s)!\n", Thread.currentThread().getName());
40       System.out.printf("----> Missao da Thread (%s) cumprida em %.2fs!\n\n", Thread.currentThread().getName(), totalSecs);
41    }
42 }
43
```



```
Missao: Gerar um bilhao de numeros aleatorios usando Threads!
-> Thread iniciada (main)...
-> Criando +1 Thread(s) para isso:
----> Thread criada (Thread-0)!
-> Thread finalizada (main) em 0,00s!
----> Iniciando Thread (Thread-0)...
----> Encerrando Thread (Thread-0)!
----> Missao da Thread (Thread-0) cumprida em 36,34s!
```

Criando Threads em Java

Exemplo 8

- Gerar 1 bilhão de números aleatórios com várias *Threads*:

  - Utilizando o console: *java MultipleTreadExample.java*

```
D:\Work Files\Meus Documentos - Robson\IMT - Grad\2023\2oSem\ECM251-Linguagem de Programação I\
Cliente-Servidor Java Sockets e Threads\Código\Projeto4>java MultipleThreadExample.java

Missao: Gerar um bilhao de numeros aleatorios usando Threads!
-> Thread iniciada (main)...
-> Criando +1 Thread(s) para isso:
----> Thread criada (Thread-0)!
-> Thread finalizada (main) em 0,00s!
----> Iniciando Thread (Thread-0)...
----> Encerrando Thread (Thread-0)!
----> Missao da Thread (Thread-0) cumprida em 39,46s!
```

Criando Threads em Java

Exemplo 9

- Gerar 1 bilhão de números aleatórios com várias *Threads*:

  - Utilizando o console: *java MultipleTreadExample.java 1*

```
D:\Work Files\Meus Documentos - Robson\IMT - Grad\2023\2oSem\ECM251-Linguagem de Programação I\
Cliente-Servidor Java Sockets e Threads\Código\Projeto4>java MultipleThreadExample.java 1

Missao: Gerar um bilhao de numeros aleatorios usando Threads!
-> Thread iniciada (main)...
-> Criando +1 Thread(s) para isso:
----> Thread criada (Thread-0)!
-> Thread finalizada (main) em 0,00s!
----> Iniciando Thread (Thread-0)...
----> Encerrando Thread (Thread-0)!
----> Missao da Thread (Thread-0) cumprida em 37,39s!
```

Criando Threads em Java

Exemplo 10

- Gerar 1 bilhão de números aleatórios com várias *Threads*:

  - Utilizando o console: *java MultipleTreadExample.java 2*

```
D:\Work Files\Meus Documentos - Robson\IMT - Grad\2023\2oSem\ECM251-Linguagem de Programação I\
Cliente-Servidor Java Sockets e Threads\Código\Projeto4>java MultipleThreadExample.java 2

Missao: Gerar um bilhao de numeros aleatorios usando Threads!
-> Thread iniciada (main)...
-> Criando +2 Thread(s) para isso:
----> Thread criada (Thread-0)!
----> Thread criada (Thread-1)!
----> Iniciando Thread (Thread-0)...
-> Thread finalizada (main) em 0,00s!
----> Iniciando Thread (Thread-1)...
----> Encerrando Thread (Thread-0)!
----> Missao da Thread (Thread-0) cumprida em 20,60s!

----> Encerrando Thread (Thread-1)!
----> Missao da Thread (Thread-1) cumprida em 20,81s!
```

Criando Threads em Java

Exemplo 11

- Gerar 1 bilhão de números aleatórios com várias *Threads*:

  - Utilizando o console: *java MultipleTreadExample.java 3*

```
D:\Work Files\Meus Documentos - Robson\IMT - Grad\2023\2oSem\ECM251-Linguagem de Programação I\E
Cliente-Servidor Java Sockets e Threads\Código\Projeto4>java MultipleThreadExample.java 3

Missao: Gerar um bilhao de numeros aleatorios usando Threads!
-> Thread iniciada (main)...
-> Criando +3 Thread(s) para isso:
----> Thread criada (Thread-0)!
----> Thread criada (Thread-1)!
----> Iniciando Thread (Thread-0)...
----> Thread criada (Thread-2)!
----> Iniciando Thread (Thread-1)...
-> Thread finalizada (main) em 0,00s!
----> Iniciando Thread (Thread-2)...
----> Encerrando Thread (Thread-0)!
----> Missao da Thread (Thread-0) cumprida em 13,53s!

----> Encerrando Thread (Thread-2)!
----> Missao da Thread (Thread-2) cumprida em 13,57s!

----> Encerrando Thread (Thread-1)!
----> Missao da Thread (Thread-1) cumprida em 13,70s!
```

Conclusões

- É fundamental entender os conceitos de programação com ***Threads*** concorrentes para criá-las e gerenciá-las de forma segura e eficiente, usando as classes e recursos fornecidos pelo **Java**, como os pacotes *java.lang.Thread* e *java.util.concurrent*;
- O uso incorreto de ***Threads*** pode levar a problemas de concorrência, como condições de corrida e bloqueios;
- Novas abordagens, como programação assíncrona e paralela, também estão disponíveis no **Java**, oferecendo alternativas às ***Threads*** tradicionais, visando melhorar o desempenho e a eficiência do código.

Tópico

- Um único Cliente para o Servidor

Um único Cliente para o Servidor

Definição

- As soluções de arquitetura **Cliente-Servidor** apresentadas anteriormente, com complexidade e dificuldade de forma iterativa e incremental por motivos didáticos, apresentavam um **Servidor** cuidando da conexão com um único **Cliente**;
- Em **Java**, pelo fato do método ***nextLine()*** ser utilizado de forma **Bloqueante**, aguardando a chegada de uma linha pelo seu fluxo de entrada, a execução do programa do **Servidor** não tem como tratar outro **Cliente** ao mesmo tempo, senão aquele que está monitorando sua entrada;

Um único Cliente para o Servidor

Exemplo (aula passada)

- Método *nextLine( )* do **Servidor** utilizado de forma **Bloqueante**:

```
56    private static void conversaComCliente() throws IOException
57    {  String msg;
58       // Escutando as mensagens do cliente que chegam ao servidor
59       while(entrada.hasNextLine())  // Aguarda proxima mensagem do cliente...
60       {  msg = leMensagemCliente();       // Le mensagem do Cliente
61          retornaMensagemCliente(msg);      // Servidor retorna mensagem ao cliente
62       }
63    }
64
65    private static String leMensagemCliente() throws IOException
66    {  String msg = entrada.nextLine();  // Le mensagem do Cliente
67       System.out.print("Chegou do Cliente: ");
68       System.out.println(msg);    // Imprime na tela a mensagem de entrada
69       return msg;
70    }
71
72    private static void retornaMensagemCliente(String msg) throws IOException
73    {  // Servidor retorna mensagem ao Cliente
74       saida.println(msg);
75       System.out.print("Ecoou ao Cliente: ");
76       System.out.println(msg);    // Imprime na tela a mensagem de saÃ-da
77    }
```

Método Bloqueante!!!

Conclusão

- Seguindo essa linha de solução, no caso de haver a necessidade de mais de um **Cliente** tentando se conectar ao mesmo **Servidor** e da implementação desse **Servidor** estar utilizando uma API de *Sockets* com operação bloqueante, como foi o caso, uma das soluções possíveis é que o **Servidor** seja implementado utilizando ***Threads*** separadas, cada uma cuidando da comunicação com um **Cliente** distinto.

Tópico

- Cliente-Servidor Java *Sockets* e *Threads*

Características

- Os códigos apresentados são aplicações simples da arquitetura **cliente-servidor** em **Java**;
- O código do **servidor** não apresenta interface gráfica, apenas mensagens de ***status***, via console;
- O código do **cliente** apresenta interface com o usuário simples, do tipo ***Swing***, além de apresentar mensagens de ***status*** via console, também;
- Essas aplicações não permitem o envio de mensagens do **servidor** para o **cliente**, somente do **cliente** para o **servidor**;
- Essas aplicações permitem conexões de mais de um **cliente** ao **servidor** por vez;

Características

- Pelo fato destas aplicações **Java** apresentarem métodos **bloqueantes** para realizar a comunicação entre **cliente-servidor**, o **servidor** precisa se dedicar a esperar por dados de um único **cliente** por vez, fazendo com que, naturalmente, a execução de sua tarefa de comunicação seja limitada a esse único **cliente**;
- Uma das estratégias, então, que possibilita ao **servidor** a conexão de mais de um **cliente** por vez, é a da criação de outras **tarefas concorrentes** à primeira, ou ***Threads***, executadas ao mesmo tempo e independentes, cada uma cuidando da conexão com um dos seus **clientes** especificamente.

Um único Cliente para o Servidor

Exemplo 12

- Projeto Java Multicliente-Servidor com *Socket* e *Threads*:

Classe *Server.java*

Um único Cliente para o Servidor

Exemplo 12

- Projeto Java Multicliente-Servidor com *Socket* e *Threads*:

```
1  import java.io.IOException;
2  import java.net.ServerSocket;
3
4  public class Server
5  {  public static final String ADDRESS = "127.0.0.1";  // IP Address local do servidor
6     public static final int PORT = 4000; //ou 3334
7     private ServerSocket serverSocket;
8
9     public void start() throws IOException
10    {  serverSocket = new ServerSocket(PORT);
11       System.out.println("Servidor iniciado na porta: " + PORT);
12       clientConnectionLoop();
13    }
14
15    private void clientConnectionLoop() throws IOException
16    {  System.out.println("Aguardando conexao de um cliente!");
17       do
18       {  ClientSocket clientSocket = new ClientSocket(serverSocket.accept());
19          new Thread(() -> clientMessageLoop(clientSocket)).start();  // Criando uma ExpressÃ£o Lambda
20       }while(true);
21    }
22
```

Um único Cliente para o Servidor

Exemplo 12

- Projeto Java Multicliente-Servidor com *Socket* e *Threads*:

```
23    public void clientMessageLoop(ClientSocket clientSocket)
24    {  String msg;
25       try
26       {  while((msg = clientSocket.getMessage()) != null && !msg.equalsIgnoreCase("sair"))
27          {  System.out.printf("Mensagem recebida do cliente %s: %s\n", clientSocket.getRemoteSocketAddress(), msg);
28          }
29       }
30       finally
31       {  clientSocket.close();
32       }
33    }
34
35    public static void main(String args[])
36    {  System.out.println("*v*v*v* CONSOLE DO SERVIDOR *v*v*v*");
37       try
38       {  Server server = new Server();
39          server.start();
40       }
41       catch(IOException ex)
42       {  System.out.println("Erro ao iniciar o servidor: " + ex.getMessage());
43       }
44       System.out.println("Servidor finalizado!");
45    }
46 }
47
```

Um único Cliente para o Servidor

Exemplo 13

- Projeto Java Multicliente-Servidor com *Socket* e *Threads*:

Classe *Client.java*

Um único Cliente para o Servidor

Exemplo 13

- Projeto Java Multicliente-Servidor com *Socket* e *Threads*:

```
1  import java.io.IOException;
2  import java.io.PrintWriter;
3  import java.net.Socket;
4  import java.util.Scanner;
5
6  public class Client
7  {  private Socket clientSocket;
8     private Scanner scanner;
9     private PrintWriter saida;
10
11    public Client()
12    {  scanner = new Scanner(System.in);
13    }
14
15    public void start() throws IOException
16    {  clientSocket = new Socket(Server.ADDRESS, Server.PORT);
17       saida = new PrintWriter(clientSocket.getOutputStream(), true);
18       System.out.println("Cliente " + Server.ADDRESS + ":" + Server.PORT + " conectado ao servidor!");
19       messageLoop();
20    }
21
```

Um único Cliente para o Servidor

Exemplo 13

- Projeto Java Multicliente-Servidor com *Socket* e *Threads*:

```
22    private void messageLoop() throws IOException
23    {  String msg;
24       System.out.println("Aguardando a digitaÃ§Ã£o de uma mensagem!");
25       do
26       {  System.out.print("Digite uma mensagem (ou <sair> para finalizar): ");
27          msg = scanner.nextLine();
28          saida.println(msg);
29       }while(!msg.equalsIgnoreCase("sair"));
30    }
31
32    public static void main(String args[])
33    {  System.out.println("*v*v*v* CONSOLE DO CLIENTE *v*v*v*");
34       try
35       {  Client client = new Client();
36          client.start();
37       }
38       catch(IOException ex)
39       {  System.out.println("Erro ao iniciar o cliente: " + ex.getMessage());
40       }
41       System.out.println("Cliente finalizado!");
42    }
43 }
```

Um único Cliente para o Servidor

Exemplo 14

- Projeto Java Multicliente-Servidor com *Socket* e *Threads*:

Classe *ClientSocket.java*

Um único Cliente para o Servidor

Exemplo 14

- Projeto Java Multicliente-Servidor com *Socket* e *Threads*:

```
1  import java.io.*;
2  import java.net.Socket;
3  import java.net.SocketAddress;
4
5  public class ClientSocket
6  {   private final Socket socket;
7      private final BufferedReader entrada;
8      private final PrintWriter saida;
9
10     public ClientSocket(final Socket socket) throws IOException
11     {   this.socket = socket;
12         System.out.println("Cliente " + socket.getRemoteSocketAddress() + " se conectou!");
13         entrada = new BufferedReader(new InputStreamReader(socket.getInputStream()));
14         saida = new PrintWriter(socket.getOutputStream(), true);
15     }
16
17     public SocketAddress getRemoteSocketAddress()
18     {   return socket.getRemoteSocketAddress();
19     }
20
```

Um único Cliente para o Servidor

Exemplo 14

- Projeto Java Multicliente-Servidor com *Socket* e *Threads*:

```
21      public void close()
22      {   try
23          {   entrada.close();
24              saida.close();
25              socket.close();
26          }
27          catch(IOException ex)
28          {   System.out.println("Erro o fechar o socket: " + ex.getMessage());
29          }
30      }
31
32      public String getMessage()
33      {   try
34          {   return entrada.readLine();
35          }
36          catch(IOException ex)
37          {   return null;
38          }
39      }
40
41      public boolean sendMsg(String msg)
42      {   saida.println(msg);
43          return !saida.checkError();
44      }
45  }
```

Conclusões

- Usar ***Threads*** no servidor permite que ele atenda a vários clientes simultânea e independentemente;
- O uso de aplicações em **Java,** com **múltiplos clientes** simultâneos conectados a um servidor, pode ser uma escolha viável para arquitetura **cliente-servidor**, dependendo dos requisitos do projeto e das necessidades específicas;
- A possibilidade de vários **clientes** se conectarem a um único **servidor**, ao mesmo tempo, abre um número grande de aplicações, uso e possibilidades de comunicação entre **cliente-servidor**.

Conclusões

- Essa solução com um **Servidor** utilizando múltiplas ***Threads***, uma para cada **Cliente** conectado a ele, é parcialmente escalável, pois sempre haverá um limite de ***Threads*** permitidas pelo Sistema Operacional – SO e, consequentemente, um limite de **Clientes** conectados a esse **Servidor** simultaneamente, sendo, mesmo assim, bastante adequada para as aplicações onde sabe-se que não será atingido esse limite de conexões.

Exercícios

1. No tópico **Expressão Lambda em Java**, completar, viabilizar, executar e registrar o funcionamento dos códigos parciais apresentados nos Exemplos 1, 2 e 3 da página 5;
2. No tópico **Funções Anônimas em Java**, executar e registrar o funcionamento dos códigos apresentados nos Exemplos 4 e 5 das páginas 9 e 10, respectivamente;
3. No tópico **Criando *Threads* em Java**, executar e registrar o funcionamento dos códigos apresentados nos Exemplos 6, 7, 8, 9, 10 e 11, das páginas 23 até 30, respectivamente;
4. O que acontece se na linha de comando digitada via console: *java MultipleTreadExample.java* ***N***, o valor de ***N*** for aumentando? Há um limite pragmático para ***N***? Justifique!

Exercícios

5. No tópico **Cliente-Servidor Java *Sockets* e *Threads***, executar e registrar o funcionamento dos códigos apresentados nos Exemplos 12, 13 e 14, das páginas 39 até 47, respectivamente;
6. O que acontece se nas aplicações dos Exemplos 12, 13 e 14, o número de **Clientes** instanciados for elevado? Há um limite para a quantidade de **Clientes** na aplicação do **Servidor** fornecida? Justifique todas as respostas!

Exercício 7

a. Criar um projeto denominado ***ProjetoMultiClientServer*** na IDE de sua preferência e digitar as classes fornecidas ***ClientSocket.java***, ***Server.java*** e ***Client.java***;

b. Executar, somente, a classe ***Client.java***, verificar e registrar o que ocorre. Explique, em detalhes, o ocorrido;

c. Executar a classe ***Server.java***, verificar e registrar o que ocorre. Explique, em detalhes, o ocorrido;

d. Sem encerrar a execução da classe ***Server.java***, executar, a seguir, a primeira instância da classe ***Client.java***, digitando algumas mensagens no seu respectivo console, uma após a outra e, por fim, optar por sair. Verificar e registrar o que ocorre. Explique, em detalhes, o ocorrido;

Exercício 7

e. Sem encerrar a execução da classe ***Server.java***, executar, a seguir, novamente, a primeira instância da classe ***Client.java***, digitando algumas mensagens no seu respectivo console, uma após a outra e não optar por sair, ainda. Verificar e registrar o que ocorre. Explique, em detalhes, o ocorrido;

f. Sem encerrar a execução da classe ***Server.java***, executar, a seguir, a segunda instância da classe ***Client.java***, digitando algumas mensagens no seu respectivo console, uma após a outra e não optar por sair, ainda. Verificar e registrar o que ocorre. Explique, em detalhes, o ocorrido;

Exercício 7

g. Sem encerrar a execução da classe ***Server.java***, executar, a seguir, a terceira instância da classe ***Client.java***, digitando algumas mensagens no seu respectivo console, uma após a outra e não optar por sair, ainda. Verificar e registrar o que ocorre. Explique, em detalhes, o ocorrido;

h. Sem encerrar a execução das classe ***Server.java*** e das classes ***Client.java*** instâncias **1**, **2** e **3**, digitar outra rodada de mensagens nos respectivos consoles das classes ***Client.java*** instâncias **1**, **2** e **3**, uma após a outra e não optar por sair, ainda, de nenhuma delas. Verificar e registrar o que ocorre. Explique, em detalhes, o ocorrido;

Exercício 7

i. Encerrar a execução das classes ***Client.java*** instâncias **1**, **2** e **3**, digitar, digitando <***sair***> no respectivo console, para cada uma delas, na ordem dada. Verificar e registrar o que ocorre. Explique, em detalhes, o ocorrido;

j. Sem encerrar a execução da classe ***Server.java***, executar, a seguir, uma segunda instância da classe ***Server.java***. Verificar e registrar o que ocorre. Explique, em detalhes, o ocorrido;

Exercício Desafio 1

- Estudar, pesquisar, desenvolver e testar novas classes ***Server2.java***, ***Client2.java*** e ***ClientSocket2.java***, baseadas nas respectivas classes fornecidas, fazendo com que a nova classe **Servidor** encerre sua execução após o último **Cliente** se desconectar da mesma.

Exercício Desafio 2

- Estudar, pesquisar, desenvolver e testar novas classes ***ServerBidirectional.java***, ***ClientBidirectional.java*** e ***ClientSocketBidirectional.java***, baseadas nas respectivas classes fornecidas, fazendo com que a nova classe **servidor** retransmita de volta para a nova classe **cliente**, as mensagens que a classe **servidor** receber dessa classe **cliente**, apresentando-as, também, em ambos os consoles.

Bibliografia Básica

- MILETTO, Evandro M.; BERTAGNOLLI, Silvia de Castro. Desenvolvimento de software II: introdução ao desenvolvimento web com HTML, CSS, javascript e PHP (Tekne). Porto Alegre: Bookman, 2014. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/9788582601969
- WINDER, Russel; GRAHAM, Roberts. Desenvolvendo Software em Java, 3ª edição. Rio de Janeiro: LTC, 2009. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/978-85-216-1994-9
- DEITEL, Paul; DEITEL, Harvey. Java: how to program early objects. Hoboken, N. J: Pearson, c2018. 1234 p. ISBN 9780134743356.

Continua...

Bibliografia Básica (*continuação*)

- HORSTMANN, Cay S; CORNELL, Gary. Core Java. SCHAFRANSKI, Carlos (Trad.), FURMANKIEWICZ, Edson (Trad.). 8. ed. São Paulo: Pearson, 2010. v. 1. 383 p. ISBN 9788576053576.
- LIANG, Y. Daniel. Introduction to Java: programming and data structures comprehensive version. 11. ed. New York: Pearson, c2015. 1210 p. ISBN 9780134670942.
- TURINI, Rodrigo. Desbravando Java e orientação a objetos: um guia para o iniciante da linguagem. São Paulo: Casa do Código, [2017]. 222 p. (Caelum).

Bibliografia Complementar

- HORSTMANN, Cay. Conceitos de Computação com Java. Porto Alegre: Bookman, 2009. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/9788577804078
- MACHADO, Rodrigo P.; FRANCO, Márcia H. I.; BERTAGNOLLI, Silvia de Castro. Desenvolvimento de software III: programação de sistemas web orientada a objetos em java (Tekne). Porto Alegre: Bookman, 2016. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/9788582603710
- BARRY, Paul. Use a cabeça! Python. Rio de Janeiro: Alta Books, 2012. 458 p.
  ISBN 9788576087434.

Continua...

Bibliografia Complementar (continuação)

- LECHETA, Ricardo R. Web Services RESTful: aprenda a criar Web Services RESTfulem Java na nuvem do Google. São Paulo: Novatec, c2015. 431 p.
  ISBN 9788575224540.
- SILVA, Maurício Samy. JQuery: a biblioteca do programador. 3. ed. rev. e ampl. São Paulo: Novatec, 2014. 544 p.
  ISBN 9788575223871.
- SUMMERFIELD, Mark. Programação em Python 3: uma introdução completa à linguagem Phython. Rio de Janeiro: Alta Books, 2012. 506 p.
  ISBN 9788576083849.

Continua...

Bibliografia Complementar (continuação)

- YING, Bai. Practical database programming with Java. New Jersey: John Wiley & Sons, c2011. 918 p.
- ZAKAS, Nicholas C. The principles of object-oriented JavaScript. San Francisco, CA: No Starch Press, c2014. 97 p. ISBN 9781593275402.
- TANENBAUM, Andrew S.; MAARTEN, V. S. Sistemas Distribuídos: princípios e paradigmas, Pearson Education. 2ª edição. 2008.
- GOETZ et. al. Java Concurrency in Practice, 1st edition, 2006.
- CALVETTI, Robson. Programação Orientada a Objetos com Java. Material de aula, São Paulo, 2020.

Aula 22 – L1/1, L2/1 e L3/1

FIM

Engenharia da Computação – 3ª série

Cliente-Servidor Java Sockets e Threads
(L1/2, L2/2 e L3/2)

2024

Aula 22 – L1/2, L2/2 e L3/2

Horário

Terça-feira: 2 x 2 aulas/semana

- L1/1 (07h40min-09h20min): *Prof. Calvetti*;
- L1/2 (09h30min-11h10min): *Prof. Calvetti*;
- L2/1 (07h40min-09h20min): *Prof. Evandro*;
- L2/2 (11h20min-13h00min): *Prof. Calvetti*;
- L3/1 (09h30min-11h10min): *Prof. Evandro*;
- L3/2 (11h20min-13h00min): *Prof. Evandro.*

Exercícios

- Terminar, entregar e apresentar ao professor para avaliação, os exercícios propostos na aula de teoria, deste material.

Bibliografia (apoio)

- LOPES, ANITA. GARCIA, GUTO. Introdução à Programação: 500 algoritmos resolvidos. Rio de Janeiro: Elsevier, 2002.
- DEITEL, P. DEITEL, H. Java: como programar. 8 Ed. São Paulo: Prentice-Hall (Pearson), 2010;
- BARNES, David J.; KÖLLING, Michael. Programação orientada a objetos com Java: uma introdução prática usando o BlueJ . 4. ed. São Paulo: Pearson Prentice Hall, 2009.

Aula 22 – L1/2, L2/2 e L3/2

FIM