INSTITUTO FEDERAL Bahia Campus Vitória da Conquista	**LISTA DE EXERCÍCIOS 02**	
	CURSO: Bacharelado em Sistemas de Informação	**MODALIDADE:** Ensino Superior
	MÓDULO/SEMESTRE/SÉRIE: 3º	**PERÍODO LETIVO:** 2024.1
	DISCIPLINA: Linguagem de Programação II	**CLASSE:** 20241.3.119.1N
	DOCENTE: Alexandro dos Santos Silva	

INSTRUÇÕES

- Para resolução das questões abaixo, será admitido o uso apenas da sintaxe adotada para escrita de programas em Java.

1. A classe que se segue abaixo reproduz o comportamento de um cronômetro cuja contagem é atualizada a cada 1.000 milissegundos (1 segundo), sem que haja parada dessa contagem. Para tal, a referida classe herda outra classe, **`Thread`**.

```
01  package lingprog2.lista02.questao01;
02
03  public class Cronometro extends Thread {
04
05     private int contador;
06     private boolean verbose;
07
08     public Cronometro(boolean verbose) {
09        contador = 0;
10        this.verbose = verbose;
11     }
12
13     public int getContador() {
14        return contador;
15     }
16
17     public boolean isVerbose() {
18        return verbose;
19     }
20
21     public void setVerbose(boolean verbose) {
22        this.verbose = verbose;
23     }
24
25     public void run() {
26        try {
27           while (true) {
28              sleep(1000);
29              contador++;
30              if (verbose) {
31                 System.out.println("Contagem atual: " + contador);
32              }
33           }
34        }
35        catch (InterruptedException e) {
36           e.printStackTrace();
37        }
38     }
39
40  }
```

   Crie uma classe utilitária munida de método estático para gerenciamento de 10 (dez) instâncias de classe **`Cronometro`**, sendo admitida 3 (três) operações, a saber: a) instanciação de uma nova instância seguida do início da execução da *thread* de cronometragem a ela associada; b) consulta de contagem atual de enésimo cronômetro, entre aqueles já instanciados; e c) consulta das contagem de todos os cronômetros instanciados. Um esboço do método estático segue-se abaixo.

2. Implemente uma classe que assuma o comportamento de uma *thread* para identificação e exibição, via método **`System.out.println`**, de números primos existentes em determinado intervalo numérico inteiro definido no momento da instanciação de cada objeto. Ao final da identificação e exibição dos números primos, deverá ser exibido ainda tempo em milissegundos consumido para a execução da tarefa.

   Observação: entende-se como número primo todo e qualquer número inteiro que seja divisível apenas por 1 (um) e ele próprio.

3. Escreva uma classe utilitária munida de método estático **`main`** através do qual seja fornecido um intervalo numérico inteiro; após isso, esse intervalo numérico deve ser subdividido em 5 subintervalos, cada um destes destinado a uma instância da classe implementada na questão anterior para a identificação e exibição de números primos. Considere, por exemplo, a entrada do intervalo de 0 a 500, haveria os seguintes subintervalos: a) 0 a 100; b) 101 a 200; c) 201 a 300; d) 301 a 400; e e) 401 a 500.

4. A classe que se segue abaixo implementa a interface **Contagem**, permitindo que a execução da contagem entre 0 e 10 ocorra em uma *thread* à parte da *thread* principal instanciada a partir da execução do método **main**.

```
01  package lingprog2.lista02.questao04;
02
03  public class Contagem implements Runnable {
04
05     private boolean contagemConcluida;
06
07     public Contagem() {
08        contagemConcluida = false;
09     }
10
11     public void run() {
12        int cont = 0;
13        while (cont <= 10) {
14           cont++;
15        }
16        contagemConcluida = true;
17     }
18
19     public boolean isContagemConcluida() {
20        return contagemConcluida;
21     }
22
23     public static void main(String[] args) {
24        Contagem c = new Contagem();
25
26        Thread thread = new Thread(c);
27        thread.start();
28
29        for (int i = 0; i < 10; i++) {
30           System.out.println(c.isContagemConcluida());
31        }
32     }
33
34  }
```

Qual a saída produzida pela execução do método **main**?

a) Uma sequência de 10 (dez) valores lógicos verdadeiros (**true**);
b) Uma sequência de 10 (dez) valores lógicos falsos (**false**);
c) Uma sequência alternada de valores lógicos verdadeiros (**true**) e falsos (**false**);
d) Saída indeterminável antes de sua execução.

5. Considere uma cronometragem regressiva (a iniciar de um número inteiro longo positivo qualquer e a encerrar em zero). Escreva uma classe que implemente um *thread* para a realização desta operação, considerando os seguintes aspectos:
   - Opcionalmente, é admitido, a cada decremento do contador, que haja uma pausa de alguns milissegundos, necessitando, para tal, de uma variável de controle que armazene intervalo de milissegundos de suspensão temporária do *thread* (se o valor desta variável for nulo, não haverá pausa);
   - Construtor de classe com dois parâmetros: a) número inteiro longo positivo, para fins de indicação de início da cronometragem regressiva; e b) quantidade de milissegundos para atribuição à variável de controle de pausa da cronometragem.

   Além disso, solicita-se aqui implementação de classe utilitária com a qual sejam demonstradas capacidades da classe acima mencionada. A classe utilitária deve dispor de método estático **main(String[] args)** através do qual sejam fornecidos três números inteiros longos positivos, para instanciação e execução das respectivas cronometragem regressivas. No ato da instanciação de cada cronometragem, é necessário indicar se haverá ou não pausa de 1.000 milissegundos a cada decremento de contador. Após isso, um bloco de repetição deve ser implementado para consulta da contagem atual de algum dos três cronômetros e/ou encerramento do programa. Para fins de exemplificação, no quadro abaixo é ilustrada saída de console após configuração das cronometragens regressivas em 50, 5 e 1.000 (esta última sem que haja pausa de cronometragem, ao contrário das demais), seguindo-se a isso algumas consultas de status das respectivas contagens antes do encerramento do programa, quando executado em determinada ocasião e em determinado ambiente computacional.

```
1ª Contagem Regressiva.............: 50
Pausa P/1ª Contagem Regressiva (S/N)? S
2ª Contagem Regressiva.............: 5
Pausa P/2ª Contagem Regressiva (S/N)? S
3ª Contagem Regressiva.............: 1000
Pausa P/3ª Contagem Regressiva (S/N)? N

Consultar Contador Regressivo (1/2/3) ou Encerrar (F)? 1
Contagem Atual de Contador 1 -> 48 (em atividade)

Consultar Contador Regressivo (1/2/3) ou Encerrar (F)? 2
Contagem Atual de Contador 2 -> 2 (em atividade)

Consultar Contador Regressivo (1/2/3) ou Encerrar (F)? 3
Contagem Atual de Contador 3 -> 0 (encerrada)
```

```
Consultar Contador Regressivo (1/2/3) ou Encerrar (F)? 1
Contagem Atual de Contador 1 -> 35 (em atividade)

Consultar Contador Regressivo (1/2/3) ou Encerrar (F)? F
```

6. Um *número triangular* é um número natural cujo valor é obtido pela soma de determinada quantidade dos *primeiros* números naturais. A título de exemplo, o número 10 é dado como um número triangular, justamente por seu valor ser obtido a partir da soma de 1, 2, 3 e 4; mesmo raciocínio não se aplica ao número 12, já que a soma dos 4 (quatro) primeiros números naturais é inferior a ele (1 + 2 + 3 + 4 = 10) e a soma dos 5 (cinco) primeiros naturais é superior a ele (1 + 2 + 3 + 4 +5 = 15). Escreva uma classe que implemente a interface **`java.lang.Runnable`**, para fins de identificação e exibição, via método **`System.out.println(long n)`**, de números triangulares inferiores a determinado número natural definido no momento da instanciação de cada objeto, a ser indicado na forma de parâmetro para o construtor da classe. Quando da escrita do método **`run()`**, exige-se uso de marcadores de tempo para cálculo de tempo de processamento da operação de identificação e exibição de números triangulares (aqui sugere-se o uso da classe **`java.time.LocalTime`**).

   Além disso, pede-se que seja escrita classe utilitária com a qual sejam demonstradas capacidades da classe acima mencionada. A classe utilitária deve dispor de método estático **`main(String[] args)`** através do qual seja fornecido um número inteiro longo positivo, para instanciação e execução de 3 (três) *threads* baseados na classe supracitada, para fins de geração dos mesmos números triangulares entre 1 (um) e aquele número fornecido. Ao final da execução das *threads*, deverá ser identificada qual delas consumiu menor tempo de processamento.

   Sugestão: a identificação do *thread* com menor consumo de tempo de processamento deve, obviamente, ocorrer após término da execução de todas os *threads*; a verificação da existência de algum *thread* com processamento ainda a fazer pode ser obtida da forma como se segue abaixo (assume-se aqui três objetos da classe instanciados previamente):

```
while (thread1.isAlive() || thread2.isAlive() || thread3.isAlive()) {
   // um ou mais threads em execução
}
```

7. Considere a implementação da classe abaixo, para fins de abstração de relógios digitais baseados no sistema horário de 24 horas. O método **`addSegundo()`** é responsável pela atualização do horário segundo a segundo.

```
01  package lingprog2.lista02.questao07;
02
03  public class Relogio {
04
05     private int hora;
06     private int minuto;
07     private int segundo;
08
09     public Relogio() {
10        this.hora    = 0;
11        this.minuto  = 0;
12        this.segundo = 0;
13     }
14
15     public Relogio(int hora, int minuto, int segundo) {
16        this.hora    = hora;
17        this.minuto  = minuto;
18        this.segundo = segundo;
19     }
20
21     public int getHora() {
22        return hora;
23     }
24
25     public int getMinuto() {
26        return minuto;
27     }
28
29     public int getSegundo() {
30        return segundo;
31     }
32
33     public void addSegundo() {
34        int t = hora * 24 * 60 + minuto * 60 + segundo; // conversão em total de segundos
35
36        t++;                                            // incremento de 1 (um) segundo
37
38        hora    = t / (24 * 60);
39        minuto  = t % (24 * 60) / 60;
40        segundo = t % 60;
41     }
42
43     public String toString() {
44        return (hora < 10 ? "0" : "") + hora + ":" +
```

```
45                (minuto < 10 ? "0" : "") + minuto + ":" +
46                (segundo < 10 ? "0" : "") + segundo;
47      }
48
49   }
```

Escreva uma classe, de nome **`RelogioThreadUtil`**, que simultaneamente herde a classe **`Relogio`** e implemente a interface **`java.lang.Runnable`**, de modo a permitir que a atualização do horário (segundo a segundo) possa ser executada concorrentemente com outras tarefas, pela instanciação de objetos da classe **`java.lang.Thread`** a partir de objetos instanciados dessa nova classe. Deverá ser considerado como horário de inicialização **0h 0min 0s**. A atualização deve ser encerrada ao se alcançar horário de término indicado ao construtor da classe. A cada atualização, deve ser exibido horário em formato *hh:mm:ss* acompanhado de percentual de tempo decorrido em relação ao tempo de término (a título de exemplo, caso o horário corrente seja 0h 0min 15s e o horário de término seja 0h 1min 0s, esse percentual seria de 25%).

Além disso, solicita-se que seja escrita classe utilitária com a qual sejam demonstradas capacidades da classe **`RelogioThreadUtil`**. A classe utilitária deve dispor de método estático **`main(String[] args)`** através do qual seja instanciado um objeto daquela classe (**`RelogioThreadUtil`**), com a indicação de horário de término através de operações de entrada.

8. Um *número palíndromo* é um número cujo reverso é ele próprio (ou seja, lido da esquerda para a direita ou da direita para a esquerda, tem-se o mesmo valor). A título de exemplo, o número 121 é dado como um número palíndromo, ao contrário de 123 (este último, se lido ao contrário, resulta em 321). Escreva uma classe que estenda a classe **`java.lang.Thread`** para permitir, de forma concorrente, a identificação e a exibição dos primeiros *n* números palíndromos, com tal quantidade de números palíndromos sendo indicada na forma de parâmetro para o construtor da classe. Quando da escrita do método **`run()`**, exige-se uso de marcadores de tempo para cálculo de tempo de processamento da operação de identificação e exibição de números palíndromos (aqui sugere-se o uso da classe **`java.time.LocalTime`**). Para simplificação da implementação, considere que a quantidade de números a serem gerados seja inferior ou igual a 1.000, de modo que cada palíndromo não tenha mais que 5 (cinco) dígitos.

   Além disso, solicita-se que seja escrita classe utilitária com a qual sejam demonstradas capacidades da classe acima mencionada. A classe utilitária deve dispor de método estático **`main(String[] args)`** através do qual seja fornecido um número inteiro positivo, para indicação da quantidade de números palíndromos a serem gerados; após isso, 3 (três) *threads* da classe geradora de números palíndromos devem ser instanciados e iniciados. Por fim, ao final da execução dos *threads*, deverá ser identificado qual deles consumiu mais tempo de processamento, além do percentual de consumo de tempo a mais em relação ao *thread* que demandou menor tempo de processamento. A saída gerada pelo programa deve ser semelhante àquela obtida abaixo, quando da execução dos *threads* para geração dos 50 primeiros números palíndromos (houve omissão intencional de saídas decorrentes da exibição de cada número palíndromo, bem como convém destacar que os tempos de processamento podem variar consideravelmente dependendo do ambiente computacional usado):

```
Informe Quantidade de Palíndromos: 50

Tempo de Processamento (Gerador 1)......: 8840000 ns
Tempo de Processamento (Gerador 2)......: 7381000 ns
Tempo de Processamento (Gerador 3)......: 8868000 ns
Gerador com Menor Tempo de Processamento: 2
Gerador com Maior Tempo de Processamento: 3 (20,15% acima)
```

   Sugestão: a identificação do *thread* com menor consumo de tempo de processamento deve, obviamente, ocorrer após término da execução de todos os *threads*; a verificação da existência de algum *thread* com processamento ainda a fazer pode ser obtida da forma como se segue abaixo (assume-se aqui três objetos da classe instanciados previamente):

```
while (thread1.isAlive() || thread2.isAlive() || thread3.isAlive()) {
   // uma ou mais threads em execução
}
```

9. Considere um somador de números gerados de forma aleatória. Escreva uma classe que implemente um *thread* para a realização desta operação, considerando os seguintes aspectos:
   - Números a serem gerados entre 0 (zero) e 100 (cem);
   - Pausa de 500 milissegundos após geração de cada número aleatório;
   - Construtor de classe apto ao recebimento de um único parâmetro, para indicação da quantidade de números aleatórios a serem gerados.

   Além disso, solicita-se aqui implementação de classe utilitária com a qual sejam demonstradas capacidades da classe acima mencionada. A classe utilitária deve dispor de método estático **`main(String[] args)`** através do qual sejam instanciados e executados 4 (quatro) somadores; todos deverão gerar a mesma quantidade de números aleatórios, a ser fornecida através de operação de entrada (vide classe **`java.util.Scanner`**). Após isso, um bloco de repetição deve ser implementado para consulta de somador que acumula, no momento, maior valor de soma ou, ao invés disso, para encerramento do programa. Para fins de exemplificação, no quadro abaixo é ilustrada saída de console após algumas consultas antes do encerramento do programa, quando executado em determinada ocasião e em determinado ambiente computacional.

```
Informe Quantidade: 55

Consultar Somadores ou Encerrar (S/E)? S
```

```
Somadores: S1 (175), S2 (140), S3 (121) e S4 (139)
Somador de Maior Valor Momentâneo: S2

Consultar Somadores ou Encerrar (S/E)? S
Somadores: S1 (217), S2 (237), S3 (172) e S4 (163)
Somador de Maior Valor Momentâneo: S3

Consultar Somadores ou Encerrar (S/E)? E
```

Observação: para a geração de números aleatórios, sugere-se aqui o uso da classe **`java.util.Random`** ou do método estático **`java.lang.Math.random`**.