INSTITUTO FEDERAL Bahia – Campus Vitória da Conquista	**AVALIAÇÃO**	
	CURSO: Bacharelado em Sistemas de Informação	**MODALIDADE:** Ensino Superior
	MÓDULO/SEMESTRE/SÉRIE: 3º	**PERÍODO LETIVO:** 2023.1
	DISCIPLINA: Linguagem de Programação II	**CLASSE**: 20222.3.119.1N
	DOCENTE: Alexandro dos Santos Silva	
	DISCENTE:	**CONCEITO:**

INSTRUÇÕES

- A atividade é composta de 18 (dezoito) questões, mas caberá à cada discente entrega da resolução de apenas 2 (duas) questões;
- Para consulta das questões cuja resolução deve ser entregue (após atribuição aleatória pelo uso de aplicativo de sorteio), acesse o endereço **https://drive.google.com/file/d/1EVh4DldpngHvOucFRNN1Zf9e2rSMA75C**
- **Em caso de identificação de plágio, todos os discentes envolvidos terão sua avaliação anulada, com consequente atribuição de conceito nulo**;
- Para resolução das questões, será admitido o uso apenas da sintaxe adotada para escrita de programas em Java;
- Será admitida entrega da resolução apenas através do ambiente da disciplina junto à plataforma Google Sala de Aula e em prazo estabelecido pelo docente.
- Para efeito de entrega da resolução, **todos os arquivos de extensão .java gerados devem ser compactados em um único arquivo, cujo nome deve corresponder ao próprio nome do discente**, sob pena, inclusive, de penalização.

---

1. (Peso: 1,5) Implemente uma classe, de nome **`Jogador`**, para fins de encapsulamento de dados típicos de um jogador de futebol. Sobre estes dados, são eles: a) nome; b) número; c) posição; d) em situação ou não de lesão física; e) quantidade de cartões amarelos acumulados; f) expulsão ou não em última partida da qual participou. Considere a inclusão de métodos *getter* e *setter* para todos os campos de instância da classe, bem como de **método construtor** através do qual tais campos possam ser inicializados através de parâmetros. Um último método a ser incluído, de nome **`isCondicaoJogo`**, determinará se o jogador instanciado terá condições de jogo, pelo retorno de valor booleano (**`true`** ou **`false`**), desde que obedeça, de forma simultânea, três condições, a saber:
   a) Ausência de lesão física;
   b) Quantidade de cartões amarelos acumulados inferior a 3 (três);
   c) Não registro de expulsão em última partida da qual o jogador participou.

   Por fim, ao final, inclua ainda um método estático **`main(String[] args)`** ou uma classe utilitária à parte munida desse método para demonstração das capacidades da classe **`Jogador`**. A implementação do referido método deve incluir operações de entrada de 15 (quinze) jogadores de uma equipe de futebol a serem instanciados em objetos da classe **`Jogador`**, seguindo-se a isso indicação da possibilidade ou não de formação de plantel mínimo de jogadores em condições de jogo para a realização de uma partida (usando-se, para tal, invocação do método **`isCondicaoJogo`**). De acordo com as regras estabelecidas pela *International Football Association Board*, cada equipe de futebol deve dispor de ao menos 11 (onze) jogadores em condições de jogo para a disputa de uma partida.

   Observação: para a definição da posição do jogador, considere o uso de classe de enumeração, conforme implementação que se segue abaixo:

```
01  public enum TipoPosicao {
02
03     GOLEIRO,        // goleiro
04     DEFENSOR,       // defensor
05     MEIOCAMPISTA,   // meio-campista
06     ATACANTE;       // atacante
07
08  }
```

2. (Peso: 1,5) Readeque a classe **`Jogador`** da questão anterior da forma como se segue abaixo:
   a) Readequação de construtor de tal modo que os campos de instância relativos à situação de lesão física, quantidade de cartões amarelos acumulados e expulsão em última partida com os valores, respectivamente, **`false`**, 0 (zero) e **`false`** (não exigência, portanto, de parâmetros de construção para estes atributos);
   b) Remoção de métodos setter dos campos de instância listados no item anterior;
   c) Inclusão de novos métodos públicos, conforme se segue abaixo:

Método	Descrição
`boolean aplicarCartaoAmarelo()`	Incremento, em 1 (uma) unidade, da quantidade de cartões amarelos acumulados pelo jogador **se quantitativo atual ser inferior a 3 (três)**
`boolean aplicarExpulsao()`	Atualização, **se ainda não estiver atualizado**, de campo de instância correspondente que indique que houve expulsão do jogador em última partida da qual ele participou
`boolean sofrerLesao()`	Atualização, **se ainda não estiver atualizado**, de campo de instância correspondente que indique apresentação de lesão física pelo jogador
`boolean recuperarLesao()`	Atualização, **se ainda não estiver atualizado**, de campo de instância correspondente que indique que o jogador não apresenta lesão física ou dela se

	recuperou
`boolean cumprirSuspensao()`	Atualização, **se ainda não estiverem atualizados**, de campos de instância que levam ao cumprimento de suspensão, de tal forma que: • Em caso de haver indicação de expulsão, este indicativo deve ser removido pela atribuição de valor booleano `false` ao campo de instância correspondente; • Em caso de haver indicação de acúmulo de três cartões amarelos, este indicativo deve ser removido pela atribuição de 0 (zero) cartões ao campo de instância correspondente.

Todos os métodos acima mencionados deverão retornar o valor booleano `true` em caso de haver atualização dos campos de instância manipulados pelos métodos (caso contrário, espera-se o retorno do valor booleano `false`). Por fim, ao final, implemente uma classe utilitária que disponha do método estático `main(String[] args)` para demonstração das capacidades da classe `Jogador`. A implementação do referido método deve incluir operações de entrada de 20 (vinte) jogadores de uma equipe de futebol, a serem instanciados em objetos da classe `Jogador` armazenados em um *array* e seguindo-se a isso possibilidade, a qualquer momento, do registro das operações representadas pelos métodos acima descritos em relação à determinado jogador (para tal, deverá ser indicado índice de armazenamento, no *array*, de objeto representativo do jogador desejado). Além disso, também deverá ser possível consultar, a qualquer momento, relação de nomes de jogadores com lesão ou suspensos.

3. (Peso: 1,5) Implemente uma classe, de nome `Tempo`, para fins de armazenamento de quantitativo ou duração de tempo expresso em horas, minutos e segundos, dispondo, para tal, de campos de instância para estes dados. Considere a inclusão de métodos *getter* e *setter* para todos os campos de instância, bem como dos métodos públicos cuja descrição segue-se abaixo:

Método	Descrição
`Tempo()`	Construtor sem parâmetros, com o qual o quantitativo ou duração de tempo a ser representada inicialmente na instância corresponderá à 0 (zero) horas, 0 (zero) minutos e 0 (zero) segundos
`Tempo(int h, int m, int s)`	Construtor com parâmetros cujos valores deverão ser atribuídos, respectivamente, aos campos de instância de horas, minutos e segundos
`String toString()`	Retorno de *string* representativa, em formato *hh:mm:ss*, do quantitativo ou duração de tempo indicado
`Tempo somar(Tempo t)`	Retorno de novo objeto da classe `Tempo` que armazene o quantitativo ou duração de tempo resultante da soma da duração de tempo expresso pelo objeto passado como parâmetro com a duração de tempo expresso pelo objeto a partir do qual o método é invocado (a soma, por exemplo, das durações de tempo 02h 50min 29s e 03h 40min 32s resultaria em 06h 31min 01s)

Por fim, ao final, inclua ainda um método estático `main(String[] args)` ou uma classe utilitária à parte munida desse método para demonstração das capacidades da classe `Tempo`. A implementação do referido método deve incluir operações de entrada de 5 (cinco) durações de tempo a serem instanciadas em objetos da classe `Tempo`, seguindo-se a isso soma de tais durações com o auxílio do método `somar` e exibição do objeto resultante da soma com o método `toString`.

4. (Peso: 1,5) Implemente uma classe, de nome `Circulo`, para fins de encapsulamento de apenas um campo de instância que seja referente ao raio do círculo dado que este é o único parâmetro que distingue, em termos de dimensões, um círculo de outro círculo. Além disso, inclua métodos *getter* e *setter* para tal campo de instância, bem como métodos públicos cuja descrição segue-se abaixo:

Método	Descrição
`Circulo(int raio)`	Construtor com um único parâmetro cujo valor deverá ser atribuído ao campo de instância de raio
`int getArea()`	Retorno de área do círculo representado pelo objeto, dado por $A = \pi R^2$, onde R é o raio
`int getPerimetro()`	Retorno do perímetro do círculo representado pelo objeto, dado por $P = 2\pi R$, onde R é o raio

Por fim, ao final, inclua ainda um método estático `main(String[] args)` ou uma classe utilitária à parte munida desse método para demonstração das capacidades da classe `Circulo`. A implementação do referido método deve incluir operações de entrada de raios de 5 círculos a serem instanciados em objetos da classe `Circulo`, seguindo-se a isso, pela invocação dos métodos `getArea` e `getPerimetro`, totalização das áreas dos círculos (a ser exibida) e identificação do raio do círculo de menor perímetro (a ser igualmente exibida). Em relação à constante numérica π, considere o uso da constante estática `Math.PI`.

5. (Peso: 1,5) Implemente uma classe, de nome `Filme`, para fins de encapsulamento de dados típicos de filmes. Sobre estes dados, são eles: a) título; b) duração em total de minutos; c) áudio original ou dublado; e d) com ou sem legenda. Considere a inclusão de métodos *getter* e *setter* para todos os campos de instância da classe, bem como de **método construtor** através do qual tais campos possam ser inicializados através de parâmetros. Por fim, exige-se a implementação de mais dois métodos públicos cuja descrição segue-se abaixo:

Método	Descrição

`String getDuracaoHorasMinutos()`	Retorno de duração do filme em formato de horas e minutos com base na duração do mesmo em total de minutos (a título de exemplificação, para um filme de **194 minutos**, sua duração seria expressa, portanto e nestes termos, por **3 horas e 14 minutos**)
`String getDescricao()`	Retorno de *string* descritiva do filme indicado pelo objeto com a inclusão tanto do título como da duração em horas e minutos (a título de exemplificação, para um filme intitulado de **"Titanic"** e com **194 minutos de duração**, deveria ser retornado **"O filme Titanic possui 3 horas e 14 minutos de duração"**)

Por fim, ao final, inclua ainda um método estático **`main(String[] args)`** ou uma classe utilitária à parte munida desse método para demonstração das capacidades da classe **`Filme`**. A implementação do referido método deve incluir operações de entrada de dados de 6 filmes a serem instanciados em objetos da classe **`Filme`**, seguindo-se a isso, pela invocação de métodos *getter* e do método **`getDescricao`**, exibição da descrição dos filmes com áudio original, legenda e duração superior à duração média dos filmes informados.

6. (Peso: 1,5) Implemente uma classe, de nome **`VeiculoAVenda`**, para fins de encapsulamento de dados de veículos genéricos que esteja à venda, a saber: a) tipo; b) ano de lançamento; e c) placa do veículo; e d) preço de venda. Considere a inclusão de métodos *getter* para os campos de instância da classe relativos aos dados aqui mencionados, bem como de **método construtor** através do qual tais campos possam ser inicializados através de parâmetros. Além disso, inclua método de nome **`getDescricao`**, para fins de retorno de *string* com descrição de todos os dados do veículo. Após isso, duas outras classes derivadas ou herdadas de **`VeiculoAVenda`** devem ser criadas, conforme se segue abaixo:
   - **`AutomovelAVenda`**, com inicialização de campo de instância relativo ao tipo com a *string* "Automovel";
   - **`MotocicletaAVenda`**, com inicialização de campo de instância relativo ao tipo com a *string* "Motocicleta".

   Em função da definição de valores constantes para o campo de instância relativo ao tipo de veículo, também será necessário definir **método construtor** para as classes **`AutomovelAVenda`** e **`MotocicletaAVenda`** sem que haja parâmetro para inicialização daquele campo (tipo de veículo). Por fim, ao final, implemente uma classe utilitária que disponha do método estático **`main(String[] args)`** para demonstração das capacidades das classes anteriores. A implementação do referido método deve incluir operações de entrada de dados de 5 (cinco) veículos, a serem instanciados em objetos das classes **`AutomovelAVenda`** e **`MotocicletaAVenda`** armazenados em um *array* de objetos da classe **`VeiculoAVenda`**, seguindo-se a isso exibição dos dados dos veículos com preço de venda superior ao preço médio de venda dos veículos informados (para tal, use o método **`getDescricao`** citado anteriormente, a ser fornecido pela classe **`VeiculoAVenda`**).

7. (Peso: 1,5) Considere a implementação da classe abaixo, para fins de representação de equipamentos eletrônicos em status de **ligado** ou **desligado**.

```
01  public class Equipamento {
02
03     private boolean ligado; // indicativo de equipamento ligado (true) ou desligado (false)
04
05     public Equipamento() {
06        this.ligado = false;
07     }
08
09     public boolean isLigado() {
10        return ligado;
11     }
12
13     public void ligar() {
14        setLigado(true);
15     }
16
17     public void desligar() {
18        setLigado(false);
19     }
20
21     private void setLigado(boolean ligado) {
22        this.ligado = ligado;
23     }
24
25  }
```

Construa uma segunda classe, de nome **`EquipamentoSonoro`**, que herde da classe acima e que inclua um campo de instância adicional para indicar o volume de funcionamento do equipamento em uma escala de 0 a 10, na qual cada unidade corresponderá à 10 decibéis (trata-se de unidade de medida usada para expressar a intensidade do som). A inclusão do campo de instância deverá ser acompanhada da inclusão apenas de método *getter*. Esta nova classe deve dispor de dois métodos que implementem, respectivamente, operações de redução e aumento do volume do equipamento em 1 (uma) unidade e observando-se, obviamente, que o volume esteja restrito à escala citada inicialmente. Além disso, exige-se reescrita do método **`desligar`** herdado da classe **`Equipamento`**, de modo que seja certificado que o volume do equipamento, ao desligá-lo, seja atualizado para o nível 0 (zero). Ao final, inclua ainda um método estático **`main(String[] args)`** ou uma classe utilitária à parte munida desse método para demonstração das capacidades da classe **`EquipamentoSonoro`**. A implementação do referido método deve incluir a instanciação de 6 (seis) objetos da classe **`EquipamentoSonoro`**, seguindo-se a isso, a qualquer momento,

seleção de um dos objetos instanciados e realização de uma das seguintes operações: a) ligamento; b) desligamento: c) redução de volume em 1 (uma) unidade; e d) redução de volume em 1 (uma) unidade (após o término da operação, deverá ser permitido encerrar a execução ou realizar outra operação com o mesmo objeto selecionado ou algum outro objeto). Por fim, ao encerrar a execução do método **`main`**, deverá ser exibido percentual de equipamentos ligados que estão com nível de volume prejudicial ao ser humano (para tal, considere que esse nível é superior à 80 decibéis).

Observação: quando da implementação dos métodos de redução e aumento de volume, considere que a atualização do campo de instância referente ao nível de volume não deve ocorrer em caso do equipamento se encontrar desligado (neste caso, é recomendável exibição de mensagem indicativa).

8. (Peso: 1,5) Considere a classe abaixo, pela qual são encapsulados dados de endereço e preço de imóveis.

```
01  public class Imovel {
02
03     private String endereco;
04     private double preco;
05
06     public Imovel(String endereco, double preco) {
07        this.endereco = endereco;
08        this.preco = preco;
09     }
10
11     public String getEndereco() {
12        return endereco;
13     }
14
15     public void setEndereco(String endereco) {
16        this.endereco = endereco;
17     }
18
19     public double getPreco() {
20        return preco;
21     }
22
23     public void setPreco(double preco) {
24        this.preco = preco;
25     }
26
27  }
```

Construa duas novas classes que herdem da classe acima, conforme descrições que se seguem abaixo:

- **`ImovelNovo`**, na qual seja incluído campo de instância para armazenar adicional no preço do imóvel acompanhado de respectivos métodos *getter* e *setter*, bem como construtor de inicialização deste novo campo e dos campos já herdados da classe acima.
- **`ImovelVelho`**, na qual seja incluído campo de instância para armazenar desconto do preço do imóvel acompanhado de respectivos métodos *getter* e *setter*, bem como construtor de inicialização deste novo campo e dos campos já herdados da classe acima.

Em ambas as classes acima, também é exigido aqui reescrita de método **`getPreco`** herdado da classe **`Imovel`**, de tal modo que o preço retornado considere o adicional ou o desconto aplicado ao preço original do imóvel (ou seja, seu **preço líquido**). Por fim, ao final, implemente uma classe utilitária que disponha do método estático **`main(String[] args)`** para demonstração das capacidades das classes descritas anteriormente. A implementação do referido método consiste na inclusão dos dados de 3 (três) imóveis novos e 3 (três) imóveis velhos, a serem instanciados em objetos, respectivamente, das classes **`ImovelNovo`** e **`ImovelVelho`**; após isso, proceda com a exibição dos endereços dos imóveis cujos preços líquidos sejam superiores ao preço líquido médio dos imóveis informados.

9. (Peso: 1,5) Implemente uma classe que disponha de métodos estáticos para conversão de intervalos de tempos envolvendo três unidades de tempo (minutos, horas e dias), conforme relação que se segue abaixo:

Método	Descrição
`double horasToMinutos(double h)`	Retorno de quantidade de minutos correspondentes ao intervalo de horas expresso pelo parâmetro **`h`** (lembre-se de que cada hora possui 60 minutos)
`double minutosToHoras(double m)`	Retorno de quantidade de horas correspondentes ao intervalo de minutos expresso pelo parâmetro **`m`** (lembre-se de que cada hora possui 60 minutos)
`double diasToHoras(double d)`	Retorno de quantidade de horas correspondentes ao intervalo de dias expresso pelo parâmetro **`d`** (lembre-se de que cada dia possui 24 horas)
`double horasToDias(double h)`	Retorno de quantidade de dias correspondentes ao intervalo de horas expresso pelo parâmetro **`h`** (lembre-se de que cada dia possui 24 horas)

Por fim, ao final, inclua ainda um método estático **`main(String[] args)`** ou uma classe utilitária à parte munida desse método para demonstração das capacidades da classe acima descrita. A implementação do referido método consiste na realização de operação de entrada de determinado intervalo de tempo (quantitativo e unidade de tempo) e unidade de tempo de conversão,

seguindo-se a isso identificação e exibição de intervalo de tempo convertido naquela unidade utilizando-se de um ou mais métodos estáticos descritos acima. Sobre a entrada das unidades de tempo, considere 3 (três) opções: minutos, horas e dias. A conversão, por exemplo, de um intervalo de tempo de 1.800 minutos para dias resultaria em 1,25 dias.

10. (Peso: 1,5) Considere a implementação da classe abaixo, para ilustração de operações típicas em contas mantidas por instituições bancárias.

```
01  public class Conta {
02
03     private int numero;          // número de identificação da conta
04     private double saldoAtual;   // saldo atual da conta
05     private double saldoMinimo;  // saldo mínimo da conta
06
07     public Conta(int numero, double saldoMinimo) {
08        this.numero     = numero;
09        this.saldoAtual  = 0;
10        this.saldoMinimo = saldoMinimo;
11     }
12
13     public int getNumero() {
14        return numero;
15     }
16
17     public double getSaldoAtual() {
18        return saldoAtual;
19     }
20
21     public double getSaldoMinimo() {
22        return saldoMinimo;
23     }
24
25     public void setSaldoMinimo(double saldoMinimo) {
26        this.saldoMinimo = saldoMinimo;
27     }
28
29     // realização de depósito
30     public void depositar(double deposito) {
31        saldoAtual += deposito;
32     }
33
34     // realização de saque
35     public void sacar(double saque) throws Exception {
36        // verificação de saldo futuro inferior ao saldo mínimo após saque
37        if (saldoAtual - saque < saldoMinimo)
38           throw new Exception("Saque leva a saldo inferior ao saldo mínimo...");
39
40        saldoAtual -= saque;     // atualização de saldo
41     }
42
43  }
```

Conforme ilustrado na codificação acima, há possibilidade de lançamento de exceção da classe genérica **`java.lang.Exception`**, quando da invocação de método **`sacar`**. (mais especificamente, em caso de tentativa de saque que levará a um saldo de conta inferior ao saldo mínimo estabelecido no momento de criação do objeto). Readapte tal codificação, com a implementação, inclusive, de novos classes e em termos que se seguem abaixo:

- Implementação de classe de exceção especializada, de nome **`SaqueExcecao`**, na qual conste campo de instância para armazenar valor do saque que leve à geração da exceção, bem como mensagem descritiva seja da forma "Falha ao realizar saque no valor de *x*", onde *x* deverá ser substituído pelo valor do saque;
- Readaptação de método **`sacar`** da classe **`Conta`**, pela substituição da instância de **`java.lang.Exception`** por uma instância da classe descrita no item anterior (**`SaqueExcecao`**);
- Readaptação de método **`depositar`** da classe **`Conta`**, de tal modo que ocorra lançamento de exceção em caso de tentativa de depósito cujo valor é nulo ou negativo; em relação à classe de exceção, considere a implementação de uma segunda classe de exceção especializada, de nome **`DepositoExcecao`**, na qual conste campo de instância para armazenar valor do depósito que leve à geração da exceção, bem como mensagem descritiva seja da forma "Falha ao realizar depósito no valor de *x*", onde *x* deverá ser substituído pelo valor do depósito;

11. (Peso: 1,5) Com base, novamente, na implementação da classe ilustrada na questão anterior, implemente uma classe utilitária que disponha de método estático **`main`** em que haja declaração de um *array* para armazenar referências de 10 (dez) objetos da classe **`Conta`**. Após isso, deverá ser permitido executar, a qualquer momento, alguma das seguintes operações: a) inserção de nova conta no *array*; b) listagem de contas já armazenadas no *array*, pela exibição de número e saldo atual de cada uma delas; e c) encerramento do programa. No decorrer da execução destas operações, admita o tratamento das seguintes situações, pelo lançamento de exceções:
    - Tentativa de inserção de nova conta no *array* após inserção prévia de outros 10 (dez) contas;
    - Tentativa de inserção de nova conta cujo número já pertença a alguma conta inserida anteriormente no *array*.

Observação: quando da inserção de nova conta, além do seu número de identificação e do seu saldo mínimo, deverá ser exigido valor inicial a ser depositado naquela conta (por invocação, após instanciação de objeto e entrada de tal valor, do método **`depositar`**).

12. (Peso: 1,5) Implemente uma classe que disponha de método estático **`main`** em que sejam exigidos dois números e operação aritmética a ser realizada, seguindo-se a isso, computação e exibição de resultado da operação indicada. Em relação à indicação da operação selecionada, admita o uso dos seguintes caracteres:

Operação	Caractere Indicador
Soma	+
Subtração	-
Multiplicação	x
Divisão	/

Em caso de entrada de caractere distinto daqueles acima mencionados para a indicação da operação aritmética a ser realizada, assegure que seja lançada uma exceção. Segue-se abaixo, a título de exemplo, resultado obtido com a tentativa de soma de dois números indicando-se o caractere "?" como indicador da operação a ser realizada.

```
Informe primeiro número...: 2
Informe segundo número....: 7
Informe operação (+|-|x|/): ?
Exception in thread "main" java.lang.Exception: Operador inválido!
      at lingprog2.aval01.questao05.OperacaoUtil.main(OperacaoUtil.java:27)
```

13. (Peso: 1,5) Implemente uma classe utilitária que disponha de método estático **`main`** em que seja exigida entrada de uma sequência de números inteiros separados por vírgula utilizando-se do método **`java.util.Scanner.nextLine()`**. Após isso, para tratamento numérico e soma dos números informados, considere o uso de dois métodos, a saber:
    - **`java.lang.String.split(separador)`**, através do qual é possível obter *array* de *substrings* a partir de *string* original, indicando-se como parâmetro caracteres separadores (neste caso, o símbolo de vírgula);
    - **`java.lang.Integer.parseInt(string)`**. através do qual é possível converter *string* em número inteiro correspondente.

    Em caso de entrada de sequência de números que não esteja formatada corretamente (em especial, pela possibilidade de lançamento de exceção da classe **`java.lang.NumberFormatException`**, quando da invocação do método **`java.lang.Integer.parseInt`**), assegure-se de que tal sequência seja exigida novamente até que ela esteja bem formatada. Para fins de exemplificação, segue-se abaixo resultado obtido com a entrada de algumas sequências de números não formatadas corretamente antes da entrada de uma última sequência formatada corretamente.

```
Sequência de números inteiros separados por vírgula: 2,5,a
Sequência inválida. Digite novamente!
Sequência de números inteiros separados por vírgula: 2,5a,t,u
Sequência inválida. Digite novamente!
Sequência de números inteiros separados por vírgula: 2,5,3
Soma = 10
```

14. (Peso: 1,5) Implemente uma classe utilitária que disponha de método estático **`main`** em que seja exigido um número inteiro para posterior instanciação de *array* de números reais de tamanho idêntico àquele número, seguindo-se a isso entrada de valores a serem armazenados naquela *array* e respectiva soma dos mesmos. Sabendo-se de que exceção da classe **`java.lang.NegativeArraySizeException`** é gerada em caso de tentativa de instanciação de *array* com tamanho negativo, trate-a de tal modo que, em caso de entrada de número inteiro indicativo de tamanho com valor negativo, seja exigido novo número até que ele seja 0 (zero) ou positivo. Para fins de exemplificação, segue-se abaixo resultado obtido com uma execução do método aqui proposto.

```
Informe Tamanho de Array: -5
Tamanho de array inválido. Digite novamente!
Informe Tamanho de Array: -3
Tamanho de array inválido. Digite novamente!
Informe Tamanho de Array: 3
Digite  1º número: 7
Digite  2º número: 4
Digite  3º número: 9
Soma = 20.0
```

Observação: não é admitido tratamento de entrada de número inteiro indicativo de tamanho do *array* com valor negativo utilizando-se exclusivamente de blocos de repetição (**`for`** ou **`while`**, por exemplo); ou seja, a inclusão de bloco **`try catch`** é obrigatória.

15. (Peso: 1,5) Implemente três classes da forma como se segue abaixo:
    - Exceção especializada, de nome **`SequenciaLetrasIlegalExcecao`**, para fins de indicação de existência, em determinada sequência de texto, de caracteres que não sejam letras;
    - Classe de nome **`TestadorStrings`**, na qual haja método estático, de nome **`isTextoMaiusculo`** e com parâmetro *string* para identificar se aquela sequência de texto possui apenas letras maiúsculas, de modo a retornar valor booleano indicativo desta verificação (**`true`** ou **`false`**); em caso de existência, na sequência, de caracteres que não sejam letras, cabe lançamento de exceção da classe descrita no item anterior;

- Classe utilitária, de nome **`StringsUtil`**, que disponha de método estático **`main`** na qual seja demonstradas capacidades das classes citadas anteriormente; sugere-se aqui entrada de uma simples sequência de caracteres e sua validação utilizando-se do método **`isTextoMaiusculo`** da classe **`TestadorStrings`**.

Observação: para a identificação de caracteres que não sejam letras ou caracteres maiúsculos, sugere-se invocação, aqui e respectivamente, de dois métodos estáticos da classe **`java.lang.Character`**: **`isLetter`** e **`isUpperCase`**; ambos podem receber como parâmetro um caractere (do tipo **`char`**, portanto) e devolvem um valor booleano indicando se o resultado da verificação é verdadeiro ou falso. Para a extração individual de cada caractere da *string*, considere, por sua vez, o método **`java.lang.String.charAt(i)`**, onde **`i`** representa o índice do enésimo caractere da *string* a partir da qual aquele método é invocado.

16. (Peso: 1,5) O IMC (Índice de Massa Corporal) é uma medida internacional usada para determinar se uma pessoa se encontra com peso ideal. Seu valor é determinado pela divisão da massa do indivíduo pelo quadrado de sua altura, onde a massa é expressa em quilogramas e a altura é expressa em metros, ou seja:

$$IMC = \frac{peso}{altura \times altura}$$

Pessoas com IMC igual ou superior a 18,50 e inferior a 25,00 são consideradas saudáveis; por sua vez, pessoas com IMC abaixo desta faixa de valores são identificadas como magras e pessoas com IMC acima desta faixa apresentam sobrepeso ou algum nível de obesidade. Dada a possibilidade de lançamento de exceção da classe **`java.lang.ArithmeticException`** em função da operação de divisão acima descrita, escreva uma classe utilitária que disponha de método estático **`main`** em que, a partir da entrada do peso e da altura de uma pessoa, seja identificado e exibido seu respectivo IMC, estando acompanhado do tratamento da eventual ocorrência daquela exceção. Em caso de geração da exceção, certifique-se da entrada, novamente, do peso e da altura seguido da identificação e exibição do IMC até que ela não ocorra mais. O uso exclusivo de blocos de repetição (**`for`** ou **`while`**, por exemplo) não é admitido, com o que a inclusão de bloco **`try catch`** é obrigatória.

17. (Peso: 1,5) Com base em um contexto de sorteio de um número inteiro entre 0 e 1.000 seguido de palpites para acertá-lo, implemente quatro classes da forma como se segue abaixo:
    - Exceção especializada, de nome **`PalpiteMenorExcecao`**, para fins de indicação de que determinado palpite é inferior ao número sorteado;
    - Exceção especializada, de nome **`PalpiteMaiorExcecao`**, para fins de indicação de que determinado palpite é superior ao número sorteado;
    - Classe de nome **`Sorteador`**, em cujo construtor haja geração de número aleatório entre 0 e 1.000 utilizando-se do método estático **`random`** da **`java.lang.Math`**, assim como método de nome **`palpitar`**, sem tipo de retorno e com parâmetro inteiro para identificar se aquele número é idêntico ao número aleatório gerado; em caso negativo, cabe lançamento de um objeto de uma das duas classes de exceções implementadas anteriormente (**`PalpiteMenorExcecao`** e **`PalpiteMaiorExcecao`**);
    - Classe utilitária, de nome **`PalpiteUtil`**, que disponha de método estático **`main`** na qual sejam demonstradas capacidades das classes citadas anteriormente, pela instanciação de objeto da classe **`Sorteador`** e, após isso, de entradas sucessivas de palpites até que o último palpite informado corresponda ao número inteiro gerado aleatoriamente.

18. (Peso: 1,5) Uma das mais simples e conhecidas técnicas de criptografia, passada à histórico como **Cifra de César**, consiste na substituição de cada letra do texto por outra que se apresenta no alfabeto *n posições* à frente da letra a ser substituída. Ao se adotar esta técnica considerando uma troca de três posições, a letra "A" seria substituída por "D", a letra "B" se tornaria a letra "E" e assim por diante; ao final, o texto **"Linguagem de Programação II"** seria criptografado como **"Olqjxdjhp gh Surjudpdfdr LL"**.

Com base no disposto acima, implemente uma classe que disponha de método estático munido de parâmetro *string* que represente texto para o qual será obtido e retornado texto criptografado correspondente, conforme técnica de criptografia aqui descrita. Como esta técnica não prevê a existência de outros caracteres que não sejam letras ou espaços em branco, implemente o método estático de modo que seja gerada uma exceção da classe em caso de passagem de parâmetro *string* com ocorrência de tais caracteres., a partir desta, temperatura correspondente em graus Fahrenheit; em caso de obtenção de valor inferior ao zero absoluto, objeto da classe **`ConversaoTemperaturaExcecao`** deverá ser gerado;
Após isso, inclua ainda um método estático **`main(String[] args)`** ou uma classe utilitária à parte munida desse método para realização de uma operação de entrada de texto, seguindo-se a isso chamada e exibição do valor retornado pelo método de criptografia implementado.

Observação: quando da implementação do método de criptografia, ignora a passagem de parâmetros *strings* que contenham caracteres acentuados; em relação à identificação de caracteres que não sejam letras, sugere-se aqui o uso do método estático **`java.lang.Character.isLetter`**.