INSTITUTO FEDERAL Bahia Campus Vitória da Conquista	**LISTA DE EXERCÍCIOS 01**	
	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. Implemente uma classe, de nome **`Data`**, que apresente três atributos inteiros em cada instância, para armazenamento, respectivamente, de mês, dia e ano correspondentes a uma determinada data. Em relação à implementação da classe, deve haver métodos **`get`** e **`set`** para cada atributo e um construtor para fins de inicialização desses mesmos atributos (assuma que os valores fornecidos sejam corretos). Acrescente ainda um método, de nome **`mostrarData`**, para retornar uma *string* representativa, no formato *dd/mm/aaaa*, da data armazenada (dia, mês e ano separados por barras). Ao final, inclua ainda um método estático **`main`** ou ou uma classe utilitária à parte munida desse método para demonstração das capacidades da classe implementada.

2. Ainda em relação à implementação da questão anterior, implemente um método, de nome **`getDiasTerminoAno`**, que retorne a quantidade de dias restantes para o término do ano indicado através dos valores correntes dos atributos da instância através da qual tal método será invocado.

3. Considere os métodos declarados na implementação abaixo (houve ocultação intencional do corpo dos métodos). Indique se está havendo sobrecarga ou sobrescrita dos mesmos. Em caso de sobrecarga ou sobrescrita, indique sob qual outro método ela está ocorrendo.

```
01  public class A {
02      ...
03      public A() { ... }
04      public A( int x ) { ... }
05      public void m1() { ... }
06      public void m1(int h) { ... }
07  }
08
09  public class B extends A {
10      ...
11      public B() { ... }
12      public void m1() { ... }
13      public void m1( double x, double y) { ... }
14      public void m2() { ... }
15  }
```

4. Implemente uma classe, de nome **`ContaCorrente`** e que represente uma conta corrente, contendo, para tal um atributo, de nome **`saldo`**, para armazenamento do saldo corrente. Em relação aos métodos de encapsulamento do atributo, considere apenas o metódo **`get`**. Além disso, implemente dois outros métodos, conforme se segue abaixo:
   - **`registrarDeposito(double valor)`**: sua execução deverá resultar no acréscimo, ao saldo corrente da conta, do valor de depósito (indicado ao método na forma de parâmetro), após o que é esperado retorno do valor booleano **`true`**; em caso do valor de depósito for negativo ou nulo, a operação de atualização de saldo deverá ser abortada, sendo seguida do retorno do valor booleano **`false`**;
   - **`registrarSaque(double valor)`**: sua execução deverá resultar no débito, em relação ao saldo corrente da conta, do valor de depósito (indicado ao método na forma de parâmetro) acrescido de um segundo débito, a título de tarifa de operação e correspondente à 0,5% desse valor, após o que é esperado retorno do valor booleano **`true`**; em caso do valor de depósito for negativo ou nulo ou ainda esse valor acrescido de tarifa de operação for superior ao saldo corrente da conta, a operação de atualização de saldo deverá ser abortada, sendo seguida do retorno do valor booleano **`false`**.

   Observação: ao final da implementação, inclua ainda um método estático **`main`** ou ou uma classe utilitária à parte munida desse método para demonstração das capacidades da classe implementada.

5. Implemente uma subclasse da classe implementada na questão anterior, de nome **`ContaCorrenteEspecial`**, para a qual tarifa de operação de saque corresponderá à 0,1% do valor de saque.

6. Implemente uma classe, de nome **`Voo`**, em que cada objeto representa um voo que ocorre em determinada data e em determinado horário. Cada voo possui no máximo 70 passageiros e a classe implementada permitirá controlar a ocupação dos assentos, devendo, para tal, dispor de atributo interno (um *array* de 70 valores booleanos). O acesso e a atualização do referido atributo se dará a partir dos métodos abaixo descritos:

Método	Descrição
`getProximoAssentoLivre()`	Retorno de número, entre 1 e 70, correspondente ao próximo assento livre do voo
`isAssentoLivre(int n)`	Retorno de valor booleano para indicar se o número de assento indicado na forma de

	parâmetro se encontra ou não disponível para ocupação e/ou reserva
`ocuparAssento(int n)`	Indicação de que o número de assento referenciado pelo parâmetro n passará a estar ocupado e/ou reservado, seguido do retorno de valor booleano verdadeiro; no entanto, se o parâmetro não representar um assento válido (qualquer número inferior a 1 ou superior a 70) ou este assento já estiver ocupado, o método apenas retornará o valor booleano falso, sem que haja qualquer modificação de estado do objeto
`getTotalAssentosLivres()`	Retorno da quantidade de assentos livres
`getTaxaOcupacao()`	Retorno de percentual de ocupação do voo (quantidade de assentos ocupados em relação à capacidade máxima de assentos)

Além de atributo interno para controle de assentos livres e ocupados, considere dois outros atributos cujos valores devem ser inicializados através de um construtor: número e data e horário de voo (a serem associados com os nomes de atributo **`numero`** e **`dataHorario`**, respectivamente). Ainda em relação a esse construtor, certifique-se de que o objeto seja instanciado de modo que todos os assentos estejam livres inicialmente. Em relação aos métodos de encapsulamento de tais atributos, considere apenas aqueles de acesso (**`get`**). Por fim, para a declaração do atributo **`dataHorario`**, considere o uso da classe **`java.util.Date`**.

7. Dado que a classe implementada na questão anterior herda todos os métodos definidos na classe **`java.lang.Object`**, sobrescreva um destes métodos: **`clone`**. Tal método é projetado para retornar um novo objeto da mesma classe, de modo a haver cópia de todos os valores dos atributos da instância a partir da qual ele é invocado.

8. Implemente uma classe herdeira da classe da questão anterior, de nome **`VooFlexivel`** e que permita, quando da instanciação de um objeto, definir capacidade máxima de passageiros (e, por consequência, de assentos do voo). Para tal, um novo construtor deve ser definido e que receba, além dos parâmetros já recebidos pelo construtor da superclasse, um parâmetro adicional para indicação do número de assentos do voo. Além disso, um novo método, de nome **`getTotalAssentos`**, deve ser implementado para retornar quantidade de assentos ou capacidade máxima de passageiros do voo instanciado.

   Observação: caso julgue necessário, sobrescreva métodos já definidos na superclasse.

9. Implemente uma classe para fins de encapsulamento de atributos típicos de funcionários contribuintes do Regime Geral de Previdência Social em nosso país, de nome **`Funcionario`**. Tais atributos incluem nome, sobrenome, quantidade de horas trabalhadas por mês e valor por hora trabalhada de cada funcionário, de modo a serem inicializados na **forma de passagem de parâmetros ao construtor** no momento em que novo objeto da classe for instanciado, excetuando-se a quantidade de horas trabalhadas para a qual considere 0 (zero) como valor inicial. Além dos métodos de acesso (**`get`**) de tais atributos, exige-se a inclusão de outros métodos cuja especificação se segue abaixo:

Método	Descrição
`void adicionarCargaHoraria(int h)`	Acréscimo de quantidade de horas representada pelo parâmetro **`h`** ao atributo responsável pela totalização da quantidade de horas até então trabalhadas pelo funcionário
`double getSalarioLiquido()`	Retorno de valor que indique salário do funcionário com base na quantidade de horas trabalhadas no mês e no valor por hora trabalhada e considerando, além disso, dedução de valor devido em função de alíquotas de contribuição previdênciária da forma como disposto em Portaria Ministerial MTP/ME Nº 12/2022, de 17/01/2022

Observação: de acordo com o Anexo II da Portaria Ministerial MTP/ME Nº 12/2022, de 17/01/2022, em se tratando de segurados empregados (inclusive domésticos e trabalhadores avulso), as faixas de contribuição e suas respectivas alíquotas seguem-se abaixo:

Faixa Salarial	Alíquota Progressiva (%)
Até 1.212,00	7,5
De 1,212,01 até 2.427,35	9,0
De 2.427,36 até 3.641,03	12,0
De 3.641,04 até 7.087,22	14,0

Por alíquota progressiva, entende-se por aquela que aumenta proporcionalmente na medida em que se aumenta a base de cálculo. Para um salário bruto no valor de R$ 1.500,00, por exemplo, este seria dividido em dois valores de acordo com a tabela acima: R$ 1.212,00 e R$ 288,00, com o que se aplicaria a cada um deles alíquotas, respectivamente, de 7,5% e 9,0%; ao final, haveria dedução no montante de R$ 116,82, levando a um salário líquido de R$ 1.383,18.

10. Implemente uma classe utilitária para demonstração das capacidades da classe **`Funcionario`** da questão anterior. A referida classe deve dispor de método estático **`main`** através do qual sejam implementadas operações de entrada, pelo usuário, de dados de nome, sobrenome e valor por hora trabalhada de 5 (cinco) funcionários, devendo eles serem encapsulados em objetos da classe **`Funcionario`** e seguindo-se a isso execução de alguma das seguintes operações:
    a) Acréscimo de nova quantidade de horas trabalhadas de algum dos 5 (cinco) funcionários;
    b) Exibição de salário liquido até o momento de cada um dos 5 (cinco) funcionários;
    c) Encerramento da aplicação.

Observação: em caso de seleção de alguma das operações elencadas pelos itens *a* e *b*, após executada, deverá ser permitido ao usuário selecionar novamente outra operação.

11. Implemente uma classe para fins de encapsulamento de dados de exemplares de títulos de uma típica biblioteca, de nome **`Exemplar`**. Tais dados incluem número de tombo, título da obra, autoria (um ou mais autores), habilitação ou não de exemplar para empréstimo e quantidade de empréstimos registrados, de modo a serem inicializados na **forma de passagem de parâmetros ao construtor** no momento em que novo objeto da classe for instanciado, excetuando-se a quantidade de empréstimos para a qual considere 0 (zero) como valor inicial. Além dos métodos de acesso (**`get`**) de tais atributos, exige-se a inclusão de outros métodos cuja especificação se segue abaixo:

Método	Descrição
`void habilitarEmprestimo()`	Habilitação de empréstimo de exemplar através da atualização de atributo correspondente
`void desabilitarEmprestimo()`	Desabilitação de empréstimo de exemplar através da atualização de atributo correspondente
`void adicionarEmprestimo()`	Incremento, em 1 (uma) unidade, de atributo que representa quantidade até então de empréstimos registrados do exemplar, caso ele se encontra habilitado para a realização de empréstimos

Observação: dada as características do atributo correspondente à habilitação ou não do exemplar para a realização do empréstimo, considere seu encapsulamento na forma de valor booleano.

12. Implemente uma classe utilitária para demonstração das capacidades da classe **`Exemplar`** da questão anterior, de nome **`ExemplarUtil`**. A referida classe deve dispor de método estático **`main`** através do qual sejam implementadas operações de entrada, pelo usuário, de dados de número de tombo, título, autoria e habilitação ou não para empréstimo de 6 (seis) exemplares, devendo eles serem encapsulados em objetos da classe **`Exemplar`** e seguindo-se a isso execução de alguma das seguintes operações:
    a) Habilitação de algum dos exemplares para empréstimos (considerando-se apenas aqueles se encontram, até então, desabilitados para tal);
    b) Desabilitação de algum dos exemplares para empréstimos (considerando-se apenas aqueles se encontram, até então, habilitados para tal);
    c) Registro de novo empréstimo de algum dos 6 (seis) exemplares;
    d) Exibição de total de empréstimos até então registrados considerando, para tal, todos os exemplares;
    e) Exibição de título e autoria do exemplar com maior quantidade, até então, de empréstimos (assuma a possibilidade de haver dois ou mais exemplares que se enquadrem nessa situação por possuírem a mesma quantidade de empréstimos registrados e essa quantidade representar a maior quantidade até então registrada);
    f) Encerramento da aplicação.

    Observação: em caso de seleção de alguma das operações elencadas pelos itens *a*, *b*, *c*, *d* e *e*, após executada, deverá ser permitido ao usuário selecionar novamente outra operação.

13. Implemente uma nova classe, de nome **`ExemplarDetalhado`**, que herde a classe **`Exemplar`** da primeira questão da avaliação. Soma-se a isso inclusão, em nova classe, de novo atributo cuja instância seja objeto da **`Date`** (pertencente ao pacote **`java.util`**), para fins de armazenamento de data e hora de último empréstimo, bem como sobrescrita de métodos já definidos na classe ou mesmo inclusão de novos métodos, conforme se segue abaixo:

Método	Descrição
`ExemplarDetalhado(int tombo, String titulo, String autores, boolean aptoEmprestimo)`	Construtor com parâmetros de inicialização de tombo, título, autoria e habilitação ou não para empréstimo (de forma análoga ao construtor **`Exemplar`** exigido para a superclasse na questão anterior)
`void adicionarEmprestimo()`	Incremento, caso o exemplar se encontra habilitado para a realização de empréstimos, em 1 (uma) unidade, de atributo que representa quantidade até então de empréstimos registrados do exemplar, bem como atualização de atributo de data e hora de último empréstimo com a data e hora corrente
`void adicionarEmprestimo(Date dataHora)`	Sobrescrita de método anterior, distinguindo-se dele apenas pela inclusão de parâmetro do tipo **`Date`** para indicar data e hora do empréstimo supostamente mais recente quando da chamada do método (em substituição à instanciação de objeto da classe **`Date`** no corpo do método para atualização do atributo correspondente)

Além disso, implemente uma segunda classe semelhante àquela da questão anterior (**`ExemplarUtil`**), mas com as seguintes adequações:
    a) Substituição de instâncias da classe **`Exemplar`** por instâncias da classe **`ExemplarDetalhado`**;
    b) Adequação de operação de registro de novo empréstimo, pela exigência de entrada, pelo usuário, de data e hora daquele empréstimo (supostamente o mais recente).

Observação: para a leitura de data/hora, sugere-se a utilização de método de instância disponibilizado pela classe **`SimpleDateFormat`** (pertencente ao pacote **`java.text`**). Demonstração de uso do referido método segue-se abaixo.

```
01  package lingprog2.lista01.questao13;
02
03  import java.text.SimpleDateFormat;
04  import java.util.Date;
05  import java.util.Scanner;
06
07  public class LeituraDataHora {
08
09     public static void main(String[] args) {
10        Scanner scanner          = new Scanner(System.in);
11        Date dataHora            = null;
12        // instância de classe de formatação e conversão de datas e/ou horas
13        SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yyyy hh:mm");
14
15        // entrada de data e hora
16        System.out.print("Informe data e horário [formato DD/MM/AAAA HH:MM]: ");
17        String dataHoraString = scanner.nextLine();
18
19        // tentativa de conversão para instanciação de objeto Date
20        try {
21           dataHora = formato.parse(dataHoraString);
22        }
23        // captura de exceção em caso de entrada de data/hora em formato inválido
24        catch(Exception e) {
25           System.out.println("Data/hora em formato inválido!");
26        }
27
28        System.out.println("Leitura de Data/Hora: " + dataHora);
29
30        scanner.close();
31     }
32
33  }
```

14. Implemente hierarquia de interfaces e classes, conforme se segue abaixo:
    - Interface para representação de qualquer forma geométrica, de nome **`FormaGeometrica`**, no qual constem assinaturas de métodos para cálculo de perímetro e de área da forma;
    - Classe abstrata para representação de quadriláteros que implementa a interface **`FormaGeometrica`**, de nome **`Quadrilatero`** e munida de construtor ao qual são indicados parâmetros correspondentes aos comprimentos de seus 4 (quatro) lados; pelo que, exige-se apenas implementação de método de cálculo de perímetro;
    - Classes concretas para representação de retângulos e quadrados que herdem da classe abstrata **`Quadrilatero`**, distinguindo-se desta última pelo construtor: a primeira classe, de nome **`Retangulo`**, deve dispor de dois parâmetros (comprimentos, respectivamente, da base e da altura); a segunda classe, de nome **`Quadrado`**, deve possuir como parâmetro apenas o comprimento de um dos lados, já que todos possuem, por definição, o mesmo valor;
    - Classe para representação de círculos que implementa a interface **`FormaGeometrica`**, de nome **`Circulo`** e cujo construtor possua apenas comprimento do raio como parâmetro;
    - Classe utilitária na qual conste implementação de método estático **`main`**, no qual seja indicado quantidade de formas geométricas cujas dimensões serão informadas; em seguida, para cada forma, deverá ser indicado o tipo (retângulo, quadrado ou círculo) e após isso, conforme tipo de forma, dimensões necessárias (todas as formas instanciadas deverão ser armazenadas em um *array* que armazena objetos que implementam a interface **`FormaGeometrica`**); por fim, os métodos de cálculo de perímetro e de área de cada um dos objetos instanciados deverão ser invocados, para fins de exibição dos dados por ele retornados em conjunto com suas respectivas dimensões.

    Observação: para a indicação dos dados através de operações de entrada, sugere-se aqui o uso da classe **`java.util.Scanner`**.

15. Considere a codificação da classe abaixo.

```
01  package lingprog2.lista01.questao15;
02
03  import java.util.Scanner;
04
05  public class DivisaoUtil {
06
07     public static void main(String[] args) {
08        int x, y, r;
09        Scanner scanner = new Scanner(System.in);
10
11        System.out.println("Operação de Divisão\n");
12
13        // entrada de dois números inteiros
14        System.out.print("Informe Dividendo...: ");
15        x = scanner.nextInt();
16        System.out.print("Informe Divisor.....: ");
17        y = scanner.nextInt();
```

```
18
19          r = x / y;      // divisão entre dois números inteiros informados
20
21          // exibição de resultado da divisão
22          System.out.println("\nResultado da Divisão: " + r);
23
24          scanner.close();
25      }
26  }
```

Identifique trechos problemáticos do método estático **main** que devem ser reescritos utilizando-se do tratamento de exceções oferecido pela linguagem Java. Para cada uma das exceções passíveis de serem levantadas, o tratamento adequado consiste na necessidade de alertar o usuário acerca do problema. Em relação à entrada de valores através do método **nextInt** do objeto **scanner**, certifique-se de que ela seja realizada quantas vezes for necessário para se chegar à execução da operação de divisão contida na linha 18.

16. A implementação da classe **Conta**, que se segue abaixo, é destinada à abstração de contas mantidas por instituições bancárias. Quando da invocação do construtor da classe, define-se saldo mínimo exigido para a conta.

```
01  package lingprog2.lista01.questao16;
02
03  public class Conta {
04
05      private double saldo;         // saldo corrente
06      private double saldoMinimo;  // saldo mínimo (saldo atual não pode ser inferior a este saldo)
07
08      // método construtor de inicialização de saldo mínimo
09      public Conta(double saldoMinimo) {
10          this.saldo = 0;           // inicialização de saldo atual (corrente) com 0 (zero)
11          this.saldoMinimo = saldoMinimo;
12      }
13
14      // retorno de valor atual do saldo mínimo
15      public double getSaldoMinimo() {
16          return saldoMinimo;
17      }
18
19      // atualização de valor atual do saldo mínimo
20      public void setSaldoMinimo(double saldoMinimo) {
21          this.saldoMinimo = saldoMinimo;
22      }
23
24      // retorno de valor atual do saldo corrente
25      public double getSaldo() {
26          return saldo;
27      }
28
29      // operação de registro de depósito em conta, com atualização de saldo corrente
30      public void depositar(double deposito) {
31          saldo += deposito;
32      }
33
34      // operação de registro de saque em conta, com atualização de saldo corrente
35      public void sacar(double saque) {
36          saldo -= saque;
37      }
38
39  }
```

Além disso, para demonstração das capacidades da classe **Conta**, segue-se abaixo classe utilitária munida de método estático **main**. Tal codificação, combinada com implementação da classe **Conta**, abre possibilidade do saldo corrente da conta ficar abaixo do saldo mínimo definido pelo atributo **saldoMinimo**, dependendo dos valores fornecidos através de operações de entrada de dados contidas nas instruções das linhas 13, 18 e 23.

```
01  package lingprog2.lista01.questao16;
02
03  import java.util.Scanner;
04
05  public class ContaUtil {
06
07      public static void main(String[] args) {
08          double valor;
09          Conta conta = null;
10          Scanner scanner = new Scanner(System.in);
11
12          System.out.print("Informe Saldo Mínimo: ");
13          valor = scanner.nextDouble();    // entrada de saldo mínimo de conta
14
15          conta = new Conta(valor);        // inicialização de objeto da classe Conta
16
```

```
17          System.out.print("\nInforme Depósito Inicial: ");
18          valor = scanner.nextDouble();    // entrada de valor de depósito inicial da conta
19
20          conta.depositar(valor);          // operação de depósito
21
22          System.out.print("\nInforme Saque após Depósito Inicial: ");
23          valor = scanner.nextDouble();    // entrada de valor de saque após depósito
24
25          conta.sacar(valor);              // operação de depósito
26
27          // exibição de saldo corrente de conta após operações de depósito e saque
28          System.out.println("\nSaldo Final: " + conta.getSaldo());
29
30          scanner.close();
31      }
32
33  }
```

Readapte o método **sacar** da classe **Conta** de modo a gerar uma exceção em caso da tentativa de saque em valor que, após debitado, resultará em saldo corrente inferior ao saldo mínimo definido para a conta.

17. Ainda em relação à questão anterior, readapte método estático **main** da classe **ContaUtil**, de modo a exigir que seja fornecido novamente valor de saque enquanto este valor resultar no lançamento de exceção ali implementada. A inclusão de bloco **try/catch** é obrigatória, não se admitindo, portanto, uso de bloco de seleção para verificar se saque de determinado valor resultado em saldo corrente da conta inferior ao saldo mínimo definido previamente para aquela conta.

18. Considere a interface **Tributavel** da forma conforme que se segue abaixo:

```
01  package lingprog2.lista01.questao18;
02
03  public interface Tributavel {
04
05      public double calcularTributos();
06
07  }
```

A interface **Tributavel** indica que **qualquer ativo ou bem tributável deve fornecer instrumento ou meio para cálculo de tributos, de modo que seja identificado e retornado valor totalizado de tais tributos** (vide método **calcularTributos**). Readeque a classe **Conta** da questão anterior para que esta implemente a interface **Tributavel**; ao fazê-lo, considere que as operações de depósito e de saque sejam tributadas em 1% usando-se como base de teto o próprio valor de cada operação, bem como restrições anteriores em relação ao saldo corrente ser inferior ao saldo mínimo definido para a conta. Os valores obtidos com a tributação deverão ser armazenados em campo de instância adicional a ser incluído na classe **Conta** para que seja viabilizada implementação do método **calcularTributos** da interface **Tributavel**, que consistiria no retorno, justamente, do valor de tal campo de instância, **além de que tais valores deverão ser deduzidos do saldo corrente da conta**. Por fim, ao final, também readeque a classe **ContaUtil** da questão anterior de maneira que, após instanciação do objeto da classe **Conta**, possa ser executado, a qualquer momento, qualquer número de operações de depósito ou saque (após o término da entrada da última operação, deverá ser permitido encerrar a execução do programa ou realizar outra operação de saque ou depósito com o objeto instanciado). Ao encerrar a execução do método **main**, deverá ser exibida saldo da conta representada pelo objeto instanciado, bem como totalização de tributos gerados utilizando-se da implementação, aqui discutida, do método **calcularTributos**.

19. Considere a codificação abaixo, no qual, após instanciado de um *array* de números inteiros, seus respectivos valores são listados.

```
01  int[] array = new int[] {2, 4, 6, 8, 10, 12};
02
03  for (int i = 0; i <= 12; i++) {
04      System.out.println(array[i]);
05  }
```

A execução das instruções acima codificadas resulta no lançamento de uma exceção da classe **java.lang.ArrayIndexOutOfBoundsException**, dada a tentativa de acesso, em determinada ocasião, a um índice de *array* inválido. A solução mais simples seria reescrever o bloco de repetição, de tal modo que acessos ao *array* seriam restritos ao elementos indexados entre 0 e 5. No entanto, ao invés disso, solicita-se aqui que seja implementada classe utilitária que disponha de método estático **main**, que, por sua vez, contenha o trecho de código acima devidamente readaptado, para fins de tratamento da exceção aqui mencionada. Quando do tratamento da exceção, deverá ser exibida mensagem indicativa de tentativa de acesso à índice inválido. Tão logo seja gerada e tratada a exceção, o programa deve ser encerrado de forma imediata, sem que haja tentativa de acesso de outros valores do *array*.

20. 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.

21. Considere a implementação da classe **`Data`** da forma como exigido nas questões 01 e 02, para fins de encapsulamento e métodos de processamento de datas. Readeque o método construtor da classe de tal modo que seja lançada uma exceção em caso de tentativa de instanciação de objetos cujos atributos de dia, mês e ano são inválidos nas seguintes condições:
    - Qualquer outro valor para o dia que não esteja entre 1 (um) e aquele que seja o último dia do mês;
    - Qualquer outro valor para o mês que não esteja entre 1 (um) e 12 (doze);
    - Qualquer valor para o ano que seja negativo ou que contenha não mais que 4 (três) dígitos.

    Observação: exige-se implementação de classe adicional específica, de nome **`DataIncorretaException`**, para o lançamento, quando for o caso, da exceção aqui discutida.

22. A codificação que se segue abaixo consiste na instanciação de um objeto da classe **`Data`** considerando-se implementação da forma como exigido nas questões 01 e 02. Readeque-a após igual readequação da classe **`Data`** da forma como exigido na questão anterior, considerando, para tal, nova entrada de dados de dia, mês e ano até que seja efetivamente instanciado um objeto daquela classe com inicialização de valores válidos para os seus respectivos campos de instância.

```
01  package lingprog2.lista01.questao22;
02
03  import java.util.Scanner;
04  import lingprog2.lista01.questao02.Data;
05
06  public class DataUtil {
07
08     public static void main(String[] args) {
09        int d, m, a;
10        Data data;
11
12        Scanner scanner = new Scanner(System.in);
13
14        // entrada de dia, mês e ano de data
15        System.out.print("Informe Dia (1-31): ");
16        d = scanner.nextInt();
17        System.out.print("Informe Mês (1-12): ");
18        m = scanner.nextInt();
19        System.out.print("Informe Ano.......: ");
20        a = scanner.nextInt();
21
22        // instanciação de objeto da classe Data
23        data = new Data(d, m, a);
24
25        // invocação de métodos do objeto instanciado da classe Data
26        System.out.println("\nData em formato DD/MM/AAAA.............: " + data);
27        System.out.println("Quantidade de dias restantes para o ano: " +
                             data.getDiasTerminoAno());
27
28        scanner.close();
29     }
30
31  }
```

Observação: a inclusão de bloco **`try/catch`** é obrigatória, não se admitindo, portanto, uso de bloco de seleção para verificar se dados de dia, mês e ano informados resultam em uma data válida.

23. Considere implementação da classe **`Voo`** da forma como exigido nas questões 06 e 07. Aqui, exige-se readequação da referida implementação em relação a alguns métodos, a saber:
    - Inclusão de parâmetro adicional em método construtor, para fins de inicialização de quantidade de assentos do voo (ao invés de estar fixado em 70 assentos, o que exigirá readequação de instrução de inicialização de *array* de valores booleanos);
    - Em relação ao método **`isAssentoLivre`**, lançamento de exceção em caso de tentativa de reserva de assento cujo número é inferior a 0 (zero) ou superior à quantidade máxima de assentos do voo representado por objeto a partir do qual aquele método é invocado;

- No tocante ao método **`ocuparAssento`**, lançamento de exceção em caso de tentativa de reserva de assento cujo número é inferior a 0 (zero) ou superior à quantidade máxima de assentos do voo representado por objeto a partir do qual aquele método é invocado (de acordo com especificação original, valor booleano falso deveria ser retornado nestas condições).

24. Implemente uma classe utilitária que disponha de método estático **`main`** em que sejam demonstradas capacidades da classe da questão anterior, pela instanciação de novo objeto com número, data e quantidade de assentos de voo definidos através de operações de entrada. Deverá ser permitido, após isso e a qualquer momento, encerrar a aplicação ou escolher alguma das seguintes operações a serem realizadas: a) consulta de disponibilidade de determinado assento, sendo precedido pela entrada de seu respectivo número; b) reserva de determinado assento, sendo igualmente precedido pela entrada de seu respectivo número; e c) consulta de taxa de ocupação atual do voo.

    Observação: em se tratando da verificação da validade do número de assento nas operações de consulta de disponibilidade e reserva, não se admite o uso de bloco de seleção em função da previsão de lançamento de exceção da forma como descrito na questão anterior (portanto, a inclusão de bloco **`try/catch`** é obrigatória na classe utilitária aqui exigida).