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:** 2025.1
	DISCIPLINA: Linguagem de Programação II	**CLASSE:** 20251.3.119.1N
	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 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 lista02.questao01;
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 lista02.questao01;
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.

2. 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.

3. 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*.

4. 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.

5. Considere a implementação da classe **Data** da forma como se segue abaixo, para fins de encapsulamento de datas e realização de operações com elas.

```
01  package lista02.questao05;
02
03  // Classe de encapulamento de datas indicadas por dia do mês, mês e ano
04  public class Data {
05
06     private int dia;   // dia (um inteiro entre 1 e, a depender do mês e ano, 28, 29, 30 ou 31)
07     private int mes;   // mês (um inteiro entre 1 e 12)
08     private int ano;   // ano (um inteiro de 4 dígitos)
09
10     public Data(int dia, int mes, int ano) {
11        this.dia = dia;
12        this.mes = mes;
13        this.ano = ano;
14     }
```

```
15
16      public int getDia() {
17         return dia;
18      }
19
20      public void setDia(int dia) {
21         this.dia = dia;
22      }
23
24      public int getMes() {
25         return mes;
26      }
27
28      public void setMes(int mes) {
29         this.mes = mes;
30      }
31
32      public int getAno() {
33         return ano;
34      }
35
36      public void setAno(int ano) {
37         this.ano = ano;
38      }
39
40      // retorno de string representativa da data encapsulada no formado DD/MM/AAAA
41      public String mostrarData() {
42         return (dia < 10 ? "0" : "") + dia + "/" + (mes < 10 ? "0" : "") + mes + "/" + ano;
43      }
44
45      // retorno de quantidade de dias restantes para o término do ano em que se situa a data
46      public int getDiasTerminoAno() {
47         // quantidade total de dias de cada mês do ano
48         int[] qtdDias = new int[] {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
49
50         // alteração de quantidade total de dias do mês de fevereiro se data for de ano bissexto
51         if ((ano % 400 == 0) || ((ano % 4 == 0) && (ano % 100 != 0)))
52            qtdDias[1]++;
53
54         if (mes == 12) {                    // se mês da data for dezembro...
55            // dias restantes equivalente à diferença entre 31 e dia da data acrescida de 1 (um)
56            return 31 - dia + 1;
57         }
58         else {                              // caso contrário (mês anterior a dezembro)...
59            // diferença entre total de dias do mês e dia da data acrescida de 1 (um)
60            int diasRestantes = qtdDias[mes-1] - dia + 1;
61
62            // contabilização de total de dias dos meses subsequentes (até alcançar dezembro)
63            for (int i = mes + 1; i <= 12; i++)
64               diasRestantes += qtdDias[i-1]; // acréscimo de total de dias do enésimo mês
65
66            return diasRestantes;            // retorno de total de dias restantes
67         }
68      }
69
70   }
```

Readeque o método construtor da classe **`Data`** 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.

6. A codificação que se segue abaixo consiste na instanciação de um objeto da classe **`Data`**. 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 lista02.questao06;
02
03   import java.util.Scanner;
04   import lista02.questao05.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: " +
28                data.getDiasTerminoAno());
29
30          scanner.close();
31      }
32
33  }
```

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.

7. Considere 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)

Uma implementação da classe **Voo**, da forma como especificado acima, segue-se abaixo:

```
01  package lista02.questao07;
02
03  import java.util.Date;
04
05  // Encapsulamento de dados de voos e respectivas operações
06  public class Voo {
07
08     public final static int TAM = 70;   // capacidade de assentos de cada voo
09
10     // array para indicação de situação de cada assento (livre, se false, ou ocupado, se true)
11     protected boolean[] assentos;
12     protected int numero;         // número de voo
13     protected Date dataHorario;   // data e horário de voo
14
15     public Voo(int numero, Date dataHorario) {
16        // inicialização de array de valores boleanos de acordo com capacidade de assentos
17        this.assentos = new boolean[TAM];
18        this.numero = numero;
19        this.dataHorario = dataHorario;
20     }
21
22     public int getNumero() {
23        return numero;
24     }
25
26     public Date getDataHorario() {
27        return dataHorario;
```

```
28      }
29
30      // obtenção de número de próximo assento livre (entre 1 e a capacidade do voo)
31      public int getProximoAssentoLivre() {
32         // consulta de situação de assentos de acordo com valores booleanos de array
33         for (int i = 0; i < assentos.length; i++) {
34            if (!assentos[i])       // se enésimo valor for false (assento livre)...
35               return i + 1;        // retorno de número de enésimo assento
36         }
37
38         return -1;                 // indicativo de ausência de assentos livres (se for o caso)
39      }
40
41      // obtenção de quantidade total de assentos livres
42      public int getTotalAssentosLivres() {
43         int qtd = 0;               // inicialização de totalizador
44
45         // consulta de situação de assentos de acordo com valores booleanos de array
46         for (int i = 0; i < assentos.length; i++) {
47            if (!assentos[i])       // se enésimo valor for false (assento livre)...
48               qtd++;               // incremento de totalizador em 1 (um) unidade
49         }
50
51         return qtd;                // retorno de total de assentos livres
52      }
53
54      // obtenção de taxa de ocupação de assentos
55      public double getTaxaOcupacao() {
56         // obtenção de quantidade de assentos ocupados com base em assentos livres
57         int assentosOcupados = assentos.length - getTotalAssentosLivres();
58
59         // cálculo e retorno de percentual de ocupação considerando capacidade de cada voo
60         return assentosOcupados * 100.0 / assentos.length;
61      }
62
63      // retorno de true se assento indicado por parâmetro for livre (caso contrário, false)
64      public boolean isAssentoLivre(int n) {
65         // retorno de operação lógica de negação sobre enésimo valor booleano de array
66         // se enésimo valor false, assento livre, caso contrário, assento não livre
67         return !assentos[n - 1];
68      }
69
70      // reserva de assento com retorno de valor booleano de acordo com resultado da operação
71      public boolean ocuparAssento(int n) {
72         if (!isAssentoLivre(n))    // se assento não estiver livre...
73            return false;           // retorno de false (reserva de assento não efetivada)
74         else {                     // caso contrário...
75            // atualização de valor boleano em índice de array correspondente ao número de assento
76            assentos[n - 1] = true;
77            return true;            // retorno de true (reserva de assento efetivada)
78         }
79      }
80
81   }
```

Posto isto, readeque 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).

8. 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).