Gabarito da Lista de Exercícios 2

```java
package br.projecao.ed.linear;

/**
 * Implementação da estrutura de uma Pilha estática sequencial linear.
 * Uma {@link Pilha} é uma um conjunto ordenado de itens no qual novos itens
    podem ser inseridos e a partir do qual podem ser eliminados em uma
    extremidade denominada topo da pilha.
 * Por ser uma implementação estática a {@link Pilha} tem um tamanho fixo, o
    que significa que depois de instanciada a capacidade da pilha não muda.
 *
 * @author alessandro.leite
 */
public class Pilha
{

  /**
   * Elementos da {@link Pilha}.
   */
  private Object[] elementos;

  /**
   * Determina o topo da {@link Pilha}
   */
  private Integer topo;

  /**
   * Cria uma pilha vazia e com um dado tamanho.
   *
   * @param tamanho
   *            Capacidade da {@link Pilha} a ser criada. Deve ser um valor
      maior ou igual a zero.
   */
  public Pilha(int tamanho)
  {
    this.elementos = new Object[tamanho];
    this.topo = 0;
  }

  /**
   * Empilha um dado elemento se a pilha não esteja cheia.
   *
   * @param elemento
   *            Elemento a ser empilhado.
   * @return <code>true</code> se o elemento estiver sido empilhado, caso
      contrário retorna
   *         <code>false</code>.
   */
  public boolean push(Object elemento)
  {
    if (this.topo < this.elementos.length)
    {
      this.elementos[topo++] = elemento;
```

```
    }
    return false;
  }

  /**
   * Desempilha e retorna o elemento que está no topo da pilha, desde que a pilha não esteja
   * vazia.
   *
   * @return O elemento que está no topo ou <code>null</code> se a pilha estiver vazia.
   */
  public Object pop()
  {
    return isEmpty() ? null : this.elementos[topo--];
  }

  /**
   * Retorna uma nova {@link Pilha} com os elementos na ordem inversa dessa {@link Pilha}. A nova
   * {@link Pilha} tem tamanho igual a essa {@link Pilha}.
   *
   * @return Uma nova {@link Pilha} com os elementos na ordem inversa dessa {@link Pilha}.
   */
  public Pilha inverter()
  {
    Pilha inversa = new Pilha(this.elementos.length);
    inversa.topo = this.topo;

    for (int i = 0, j = this.topo - 1; i < this.topo; i++, j--)
    {
      inversa.elementos[i] = this.elementos[j];
    }
    return inversa;
  }

  /**
   * Retorna uma cópia dessa {@link Pilha}.
   *
   * @return Uma cópia dessa {@link Pilha}
   */
  public Pilha copia()
  {
    Pilha copia = new Pilha(this.elementos.length);
    copia.topo = this.topo;

    for (int i = 0; i < this.topo; i++)
    {
      copia.elementos[i] = this.elementos[i];
    }
    return copia;
  }

```

```
    /**
99   * Empilha todos os elementos de uma dada {@link Pilha} a essa {@link Pilha
         }, sem modificar o
     * estado da {@link Pilha} dada para ser concatenada com essa. Os elementos
         são empilhados até a
101  * capacidade máxima da {@link Pilha}.
     *
103  * @param p
     *            Pilha que deve ter todos os seus elementos empilhados nessa {
         @link Pilha}.
105  */
    public void concatena(Pilha p)
107 {
      // realiza uma cópia da pilha p para que ela não tenha o seu estado
          modificado.
109   Pilha copia = p.copia();

111   while (!copia.isEmpty())
      {
113     this.push(copia.pop());
      }
115 }

117 /**
     * Realiza uma pesquisa por um dado valor na {@link Pilha}, onde se o
         elemento estiver presente
119  * retorna 1 caso contrário retorna 0.
     *
121  * @param value
     *            Valor a ser pesquisado na {@link Pilha}.
123  * @return O valor 1 se o elemento estiver presente, ou 0 (zero) caso
         contrário.
     */
125 public int search(Object value)
    {
127   int i = this.topo;

129   while (i >= 0 && this.elementos[i] != value)
        i--;
131
      return i < 0 ? 0 : 1;
133 }

135 /**
     * Apresenta no console {@link System#out} os elementos da {@link Pilha}.
137  */
    public void print()
139 {
      for (int i = topo - 1; i >= 0; i--)
141   {
        System.out.println(this.elementos[i]);
143   }
    }
145
```

```
    /**
147  * Retorna a quantidade de elementos presentes na pilha, ou seja, a altura
         da {@link Pilha}. O
     * tamanho da pilha é diferente a sua capacidade. A capacidade representa a
         quantidade máxima de
149  * elementos permitidos em uma pilha, enquanto o tamanho, representa a
         quantidade de elementos
     * que foram adicionados.
151  *
     * @return
153  */
    public int size()
155 {
      return this.topo;
157 }

159 /**
     * Retorna <code>true</code> se a pilha estiver vazia ou <code>false</code>
         contrário. Um pilha
161  * é considerada vazia quando não há nenhum elemento empilhado.
     *
163  * @return <code>true</code> se a pilha estiver vazia ou <code>false</code>
         contrário.
     */
165 public boolean isEmpty()
    {
167   return this.topo == 0;
    }
169 }
```

Listing 1: Implementação de Pilha em Java

1. Implemente um algoritmo que receba duas pilhas, p_1, p_2, e passe todos os elementos da pilha p_2 para o topo da pilha p_1.

```
1   /**
     * Cria e retorna uma nova {@link Pilha} com todos dos elementos da
         pilha p1 e os elementos da
3    * pilha p2 no topo.
     *
5    * @return Uma nova {@link Pilha} com todos os elementos da {@link
         Pilha} p2 no topo da
     *         {@link Pilha} p1.
7    */
    public Pilha questao1(Pilha p1, Pilha p2)
9   {
      Pilha pilha = new Pilha(p1.size() + p2.size());
11    pilha.concatena(p1);
      pilha.concatena(p2);
13
      return pilha;
15  }
```

Listing 2: Concatena duas pilhas

2. Implemente um algoritmo que receba uma pilha e um número inteiro como parâmetros e retorne 1 se o valor informado fizer parte da pilha ou 0 (zero) caso contrário.

```
Veja o método search na classe Pilha.
```

3. Implemente um algoritmo que receba uma pilha como parâmetro e retorne como resultado uma cópia dessa pilha.

```
Veja o método inverter da classe Pilha.
```

4. Implemente um algoritmo que leia um conjunto de valores inteiros e armazene estes valores em duas pilhas, uma para os valores positivos lidos e a outra, para os valores negativos.

```
/**
 * Ler um conjunto de valores inteiros e armazena-os em duas pilhas.
    Todos os valores positivos
 * lidos são colocado na Pilha[0] e os valores negativos na Pilha[1].
 *
 * @return Duas pilhas sendo que primeira só possui valores positivos
    enquanto a segunda só
 *         possui valores negativos.
 */
public Pilha[] questao4()
{
  Pilha[] pilhas =
  { new Pilha(Integer.MAX_VALUE), new Pilha(Integer.MAX_VALUE) };

  int value;
  do
  {
    System.out.println("Informe um valor ou 0 (zero) para finalizar.")
        ;
    Scanner scanner = new Scanner(System.in);
    value = scanner.nextInt();
    if (value > 0)
    {
      pilhas[0].push(value);
    } else
    {
      pilhas[1].push(value);
    }
  } while (value != 0);

  return pilhas;
}
```

Listing 3: Separa os valores lidos em duas pilhas

5. Imagine um colecionador de vinhos que compra vinhos recentes e os guarda em uma adega para envelhecerem, e que a cada ocasião especial abre sempre a última aquisição (para poupar os mais antigos). Construa um programa que:

   - permita incluir novos vinhos na adega;
   - informe qual vinho deve ser aberto em uma ocasião especial;
   - relacione as cinco aquisições mais antigas.

   Os dados básicos que o registro de vinhos deve conter são: nome do produto e safra.

```java
package br.projecao.ed.linear.exercicios;

public class Vinho
{
  /**
   * Nome do vinho.
   */
  private String nome;

  /**
   * A safra de um vinho nada mais é do que o ano de produção do
       vinho.
   */
  private Integer safra;

  public Vinho(String nome, Integer safra)
  {
    this.nome = nome;
    this.safra = safra;
  }

  /**
   * @return the nome
   */
  public String getNome()
  {
    return nome;
  }

  /**
   * @return the safra
   */
  public Integer getSafra()
  {
    return safra;
  }
}
```

Listing 4: Código da classe Vinho

```java
package br.projecao.ed.linear.exercicios;

import br.projecao.ed.linear.Pilha;

/**
 * Uma adega de vinhos
 *
 * @author alessandro.leite
 */
public class Adega
{
  private Pilha vinhos = new Pilha(Integer.MAX_VALUE);

  /**
   * Inclui um novo {@link Vinho} na {@link Adega}
   *
   * @param vinho
   *            {@link Vinho} a ser adicionado à {@link Adega}.
   */
  public void incluir(Vinho vinho)
  {
    this.vinhos.push(vinho);
  }

  /**
   * Retorna o {@link Vinho} que deve ser aberto. O {@link Vinho} a
      ser aberto é sempre o último
   * que foi adquirido, ou seja, o {@link Vinho} adquirido
      recentemente.
   *
   * @return O {@link Vinho} que deve ser aberto.
   */
  public Vinho abrir()
  {
    return (Vinho) this.vinhos.pop();
  }

  /**
   * Retorna os cinco vinhos mais antigos da {@link Adega}.
   *
   * @return Os cinco vinhos mais antigos da {@link Adega}.
   */
  public Vinho[] obterCincoVinhosMaisAntigos()
  {
    Vinho[] aquisicoes = new Vinho[5];

    Pilha vinhos = this.vinhos.inverter();

    int i = 0;
    while (i < 5 && !vinhos.isEmpty())
    {
      aquisicoes[i++] = (Vinho) vinhos.pop();
    }
```

```
52
        return aquisicoes;
54    }
  }
```

Listing 5: Código da classe Adega

6. O problema das Torre de Hanói é bastante estudo em computação. O problema consiste em N discos de diferentes diâmetros e três estacas: A, B, C. Inicialmente os discos estão encaixados na estaca A onde o menor está em cima do maior disco. O objetivo é deslocar os discos para uma estaca C, usando a estaca B como auxiliar. Somente o primeiro disco de cada estaca pode ser deslocado. Construa a resolução desse exercício considerando $N = 4$, ou seja, para quatro discos.
7. Uma pilha pode ser usada para rastrear os tipos de escopos encontrados em uma expressão matemática e verificar se o uso deles está correto. Os delimitadores de escopo podem ser os parênteses, os colchetes e as chaves. Escreva um programa que leia uma expressão matemática e verifique se os escopos estão posicionados de forma correta.

```
1   /**
     * Verifica se os escopos dos delimitadores: parênteses (),
3    * os colchetes [] e as chaves {} estão posicionados
     * de forma correta.
5    *
     * @param expressao Expressão matemática a ser verificada se
7    *                  os escopos dos delimitadores estão posicionados
     *                  de forma correta.
9    * @return <code>true</code> se os escopos dos delimitadores estão
     *         posicionados de forma correta ou <code>false</code> caso
11   *         contrário.
     */
13  boolean balanceada(String expressao) {
      Pilha<Character> pilha = new Pilha<Character>();
15
      for (char c : expressao.toCharArray()) {
17      switch (c) {
        case '(':
19      case '[':
        case '{':
21        pilha.push(c);
          break;
23      case ')':
          if (pilha.isEmpty())
25          return false;
          else {
27          char d = pilha.pop();
            if ('(' != d)
```

```
29              return false;
            }
31          break;
        case ']':
33          if (pilha.isEmpty())
              return false;
35          else {
              char d = pilha.pop();
37            if ('[' != d)
                return false;
39          }
            break;
41      case '}':
            if (pilha.isEmpty())
43            return false;
            else {
45            char d = pilha.pop();
              if ('{' != d)
47              return false;
            }
49          break;
      }
51  }
    return pilha.isEmpty();
53 }
```

8. Escreva um programa que converta uma expressão **posfixa** na correspondente expressão **infixa**.
9. A formatação de um documentação html é definida por tags. As tags são usados em pares: um de abertura e um de fechamento. Exemplo $< em >$*itálico*$< /em >$ e $< strong >$**negrito**$< /strong >$. Escreva um programa que verifique se as tags de um arquivo html estão corretamente fechadas. Considere apenas as tags strong, tt, small, big, h1, h2, h3, h4, ul, blockquote, div, table, tr, td, address, head, body, html.
10. Simule a execução do algoritmo de conversão para a notação posfixa com a expressão aritmética abaixo:

$$(A + B) * D + E/(F + A * D) + C$$