

Started on Sunday, 27 November 2022, 11:44 PM**State** Finished**Completed on** Sunday, 27 November 2022, 11:55 PM**Time taken** 10 mins 52 secs**Grade** 16,00 out of 20,00 (80%)**Question 1**

Correct

Mark 4,00 out of 4,00

Considere o seguinte código em Java:

```
int f1() {  
    X<Integer> c = new X(1);  
    int sum = 0;  
    for(Integer elem : c) {  
        if(elem > 0)  
            sum += elem;  
    }  
    return sum;  
}
```

Identifique a **afirmação correta**:

Select one:

- ☒ a. **x** é uma classe que implementa a interface **Iterable**; ✓
- ☐ b. **x** é uma coleção *standard* do java;
- ☐ c. **c** assume o papel de *Concreteliterator* do padrão Iterator;
- ☐ d. Nenhuma está correta;

Question 2

Incorrect

Mark 0,00 out of 4,00

Considere o seguinte código parcial de uma implementação de `Stack`:

```
public class StackImpl<T> implements Stack<T> {
    private T[] elements;
    private int size;
    //...
    @Override
    public T pop() throws EmptyQueueException {
        //...
        return elements[--size];
    }
    @Override
    public Iterable<T> iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<T> {
        private int cursor;
        public MyIterator() {
            // A ?
        }
        @Override
        public boolean hasNext() {
            return // B ?
        }
        @Override
        public T next() {
            if(!hasNext()) return null;
            return // C ?
        }
    }
}
```

Se pretendermos que o iterador fornecido faça uma **travessia do topo para a base da pilha**, que opção satisfaz isto no código em falta?

Select one:

- ☐ a. **A:** `cursor = 0` | **B:** `cursor <= size - 1` | **C:** `elements[cursor++]`
- ☐ b. **A:** `cursor = 0` | **B:** `cursor < size - 1` | **C:** `elements[cursor++]`
- ☐ c. **A:** `cursor = size - 1` | **B:** `cursor >= 0` | **C:** `elements[cursor--]`
- ☒ d. **A:** `cursor = size - 1` | **B:** `cursor > 0` | **C:** `elements[cursor--]` ✖

Question 3

Correct

Mark 4,00 out of 4,00

Considere a seguinte implementação de `Tree` e o seu iterador:

```
public class TreeImpl<T> implements Tree<T> {
    private TreeNode root;
    //...
    @Override
    public Iterable<T> iterator() {
        return MyIterator();
    }

    private class MyIterator implements Iterator<T> {
        private Queue<Position<T>> queue = new QueueImpl<>();
        public MyIterator() {
            queue.enqueue(root);
        }
        @Override
        public boolean hasNext() {
            return !queue.isEmpty();
        }
        @Override
        public T next() {
            if(!hasNext()) return null;
            Position<T> p = queue.dequeue();
            for(Position<T> child : children(p)) {
                queue.enqueue(child);
            }
            return p.element();
        }
    }
}
```

Qual a travessia fornecida pelo iterador?

Select one:

- ☐ a. Travessia *em-ordem*;
- ☒ b. Travessia *breadth-first*; ✓
- ☐ c. Travessia *depth-first*;
- ☐ d. Outra;

Question 4

Correct

Mark 4,00 out of 4,00

Considere o seguinte código que utiliza o padrão *Strategy*:

```
public class BagOfWords {
    private List<String> words;
    private Sorting sort;

    public BagOfWords() {
        words = new ArrayList<>();
        sort = new SortingAscending();
    }

    public void changeSorting(Sorting s) {
        sort = s;
    }
    @Override
    public String toString() {
        sort.perform(words);
        return words.toString();
    }
    //...
}

public class Main {
    public static void main(String[] args) {
        BagOfWords bag = new BagOfWords();
        bag.put("Pattern");
        bag.put("Design");
        bag.put("Strategy");
        System.out.println(bag);
        bag.changeSorting(new SortingDescending());
        System.out.println(bag);
    }
}
```

Identifique **participantes** do padrão nesta aplicação do mesmo:

Select one:

- ☐ a. **Client:** Main | **Context:** BagOfWords | **Strategy:** SortingAscending | **Concrete Strategy:** Sorting;
- ☐ b. **Client:** BagOfWords | **Context:** Main | **Strategy:** Sorting | **Concrete Strategy:** SortingDescending;
- ☒ c. **Client:** Main | **Context:** BagOfWords | **Strategy:** Sorting | **Concrete Strategy:** SortingDescending; ✓
- ☐ d. **Client:** BagOfWords | **Context:** Main | **Strategy:** SortingAscending | **Concrete Strategy:** SortingDescending;

Question 5

Correct

Mark 4,00 out of 4,00

Considere o seguinte código que utiliza o padrão *Strategy*:

```
public class BagOfWords {
    private List<String> words;
    private Sorting sort;

    public BagOfWords() {
        words = new ArrayList<>();
        sort = new SortingAscending();
    }

    public void changeSorting(Sorting s) {
        sort = s;
    }
    @Override
    public String toString() {
        sort.perform(words);
        return words.toString();
    }
    //...
}

public class Main {
    public static void main(String[] args) {
        BagOfWords bag = new BagOfWords();
        bag.put("Pattern");
        bag.put("Design");
        bag.put("Strategy");
        bag.changeSorting(new SortingDescending());
        System.out.println(bag);
    }
}
```

Assinale a **afirmação verdadeira**:

Select one:

- ☐ a. O *output* do programa é {"Design", "Pattern", "Strategy"};
- ☒ b. É possível alterar a estratégia em *tempo de execução*; ✓
- ☐ c. A interface *Sorting* contém o método `void perform(List<Integer> l)`;
- ☐ d. Só é possível alterar a estratégia utilizada em *tempo de compilação*;