Implementando Collections e Streams com Java

> H1 - Curso; H2 - Módulo; H3 - Aula.

Collections I - List

Identificando as interfaces de Collections:

- A API java.util.Collection veio com facilidades na manipulação de Arrays;
- Principalmente no problema dos arrays serem estáticos;

O que é List e como trabalhar com ela:

java.util.List:

- Implementações abordadas:
  - java.utilArrayList;
  - java.util.Vector.
- Garante ordem de inserção
- Permite adição, atualização, leitura e remoção de arrays.
- Permite ordenação através de comparators.

Java.util.ArrayList:

Exemplos:

```
// Importando as bibliotecas necessárias
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Main {
    public static void main(String[] args) {
        // Criando uma lista com o nome de "nomes"
        List<String> nomes = new ArrayList<>();

        // Adicionando itens
        nomes.add("c");
        nomes.add("e");
        nomes.add("a");

        System.out.println(nomes);
        // Printa: [c, e, a]

        //substitui "e" por "b"
        nomes.set(1, "b");
```

```
        System.out.println(nomes);
        // Printa: [c, b, a]

        // Ordena a lista por ordem alfabética
        Collections.sort(nomes);

        System.out.println(nomes);
        // Printa: [a, b, e]

        // .remove() recebe o index ou o item
        nomes.remove(1); //remove "b"
        nomes.remove("c"); // remove "c"

        System.out.println(nomes);
        // Printa: [a]

        // .get(n) retorna o item naquele index
        String nome = nomes.get(0); //retorna "a" e insere na definição da
variável

        System.out.println(nome);
        //Printa: a

        // .size() retorna um int representando a quantidade de itens na lista
        System.out.println(nomes.size());
        // Printa: 1

        // .contains(item) retorna um boolean dizendo se aquele elemento existe
na lista
        System.out.println(nomes.contains("n"));
        // Printa: false

        // .isEmpty() retorna booleano
        System.out.println(nomes.isEmpty());
        // Printa: false

        // .clear() exclui todos os itens
        nomes.clear();

        // Agora:
        System.out.println(nomes.isEmpty());
        // Printa: true
    }
}
```

Outos métodos:

- **.indexOf(item)** - retorna int representando o index do item na lista, se o item não é encontrado, é retornado -1;

Passando pelos itens de uma lista:

Com for each:

```
...
    // Primeiro, vamos adicionar mais itens:
        nomes.add("ana");
        nomes.add("carlos");
```

```
        nomes.add("bia");
        nomes.add("kauan");
        nomes.add("luis");

    // Usamos a estrutura for no formato:
    // for (varType varName: ArrayList) {}
    // a variavel definida vai representar um da lista item em cada loop
    for (String nomeDoItem: nomes) {
        System.out.println("O nome desse é: " + nomeDoItem);
    }
    /* Printa:
        O nome desse Ã©: ana
        O nome desse Ã©: carlos
        O nome desse Ã©: bia
        O nome desse Ã©: kauan
        O nome desse Ã©: luis  */
...
```

Com Iterator object e while:

```
// Precisa-se importar:
import java.util.Iterator;

...
    // Criamos um objeto no fortamto:
    // Iterator<TypeOfItens> nameOfObjectVar = ArrayList.iterator();
    Iterator<String> nomesIterator = nomes.iterator();

    // Agora podemos usar na estrutura while nesse formato:
    while (nomesIterator.hasNext()) {
        System.out.println("O nome desse eh: " + nomesIterator.next());
    }
    /* Printa:
        O nome desse eh: ana
        O nome desse eh: carlos
        O nome desse eh: bia
        O nome desse eh: kauan
        O nome desse eh: luis  */
...
```

Collections II - Queue

O que é Queue?

- Implementação que veremos:
  - java.util.LinkedList
- Garante ordem de inserção
- Permite adição, leitura e remoção considerando a regra basica de uma fila: Primeiro que entra, primeiro que sai.
- Não permite mudança de ordenação.

Principais Métodos:

- **poll()** - retorna o próximo da fila e o exclui, retorna null no caso da fila estar vazia;
- **peek()** - retorna o próximo da fila, mas não o excui, retorna null no caso da fila estar vazia;
- **element()** - retorna o próximo da fila e não o exclui, porém retorna uma exceção caso a fila esteja vazia.

Exemplos:

```
// Importando Bibliotecas necessárias
import java.util.Queue;
import java.util.LinkedList;

public class Main {
    public static void atender(String name) {
        /* IMPLEMENTAÇÃO DE ATENDIMENTO */
        System.out.println(name + " foi atendido(a);");
    }
    public static void main(String[] args) {
        // Cria no formato:
        // Queue<ElementsType> nameOfObject = new LinkedList<>();
        Queue<String> fila = new LinkedList<>();

        // Adicionando elementos:
        fila.add("Marc");
        fila.add("Rob");
        fila.add("Bob");
        fila.add("Cub");
        fila.add("Flavy");
        fila.add("Andy");

        System.out.println("> FILA: " + fila);
        // Printa: > FILA: [Marc, Rob, Bob, Cub, Flavy, Andy]

        // Atende cada um dos elementos:
        while (fila.peek() != null) {
            atender(fila.poll());
            System.out.println("> FILA: " + fila);
        }

        // Com a lista já vazia, tenta usar fila.element e joga um erro de
exeção. Trata esse erro.
        try {
            System.out.println(fila.element());
        } catch (java.util.NoSuchElementException e) {
            System.out.println("tem nada naum");
        }
    }
}
```

Collections III - Set

Aprenda sobre as implementações de Set:

java.util.Set:

- Implementações abordadas:

    - java.util.HashSet
    - java.util.TreeSet
    - java.util.Linked
- Não garantem ordem;
- Não permite valores repetidos;
- Permite adição e remoção normalmente;
- Não possui busca por item e atualização;
- Permite navegação;
- Não permite mudança de ordenação;

Tipo	Quando utilizar	Ordenação	Performance
HashSet	Quando não é necessário manter uma ordenação	Não é ordenado e não permite repetições de valores	Tem a melhor performance
LinkedHashSet	Quando é necessário manter a ordem de inserção dos elementos	Mantém ordem de inserção	Tem a performance mais lenta
TreeSet	Quando é necessário alterar a ordem através do uso de comparators	Mantém e pode ser ordenado	Performático para leitura. Para modificação é o mais lento.

Collections IV - Map

Aprenda quando utilizar Map:

- **Não deriva de java.util.Collections**
- Implementações que veremos
    - java.util.HashMap
    - java.util.TreeMap
    - java.util.HashTable
- Entrada de chave e valor
- Permite valores repetidos, mas não chaves repetidas
- Permite adição, busca por chave ou valor, atualização, remoção e navegação
- Pode ser ordenado

Alguns Métodos:

- **put()** - tem como parâmetros, a chave e um valor. Se a chave não existe, cria. Se existem, substitui o valor.
- **get()** - tem como parâmetro uma chave, e retorna o valor.
- **containsKey()** - verifica se a chave passada como parâmetro existe ou não.
- **containsValue()** - verifica se um determinado valor existem em qualquer uma das chaves.
- **remove()** - a partir da chave;
- **size()** - retorna o tamanho do mapa;

Collections V - Comparators

Aprenda a trabalhar com a interfece Comparators

- Interfaces que aprenderemos:
  - java.util.Coparator - Interface para definir com regra de ordenação.
  - java.util.Comparable - Interface para definir regra de ordenação em uma classe de domínio.
- Algoritimos de ordenação
- Utilizando primariamente em java.util.List
- Permite a ordenação de objetos complexos

Código exemplo:

```
public class Estudante implements Comparable<Estudante>{
    private Integer age;
    private String name;

    Estudante(Integer age, String name) {
        this.name = name;
        this.age =  age;
    }

    public void setName(String name) {this.name = name;}
    public void setAge(Integer age) {this.age = age;}

    public String getName() {return this.name;}
    public Integer getAge() {return this.age;}

    @Override
    public String toString() {
        return("Estudante " + this.getName() + " - " + this.getAge());
    }

    @Override
    public int compareTo(Estudante e) {
        int n = this.getAge() - e.getAge();
        return(n);
    }

    public boolean isGreatterThan(Estudante e) {
        int n = compareTo(e);
        if (n == 0 || n < 0) {return(false);}
        else {return(true);}
    }

    public boolean isLowerThan(Estudante e) {
        int n = compareTo(e);
        if (n == 0 || n > 0) {return(false);}
        else {return(true);}
    }

    public boolean isEqualsTo(Estudante e) {
        int n = compareTo(e);
        if (n == 0) {return(true);}
```

```
        else {return(false);}
    }
}
```

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        Estudante e1 = new Estudante(15, "Luis");
        Estudante e2 = new Estudante(16, "Luiza");
        Estudante e3 = new Estudante(17, "Pedro");

        List<Estudante> en = new ArrayList<>();
        en.add(e1);
        en.add(e2);
        en.add(e3);

        System.out.println(en);

        //en.sort((first, second) -> second.getAge() - first.getAge());
        //en.sort(Comparator.comparingInt(Estudante::getAge).reversed());

        Collections.sort(en);
        System.out.println(en);
    }
}
```

Utilizando o Optional

Aprenda a criar um Optional

- Tratar de dados que podem ser nulos;
- Possui dois estado:
    - Presente;
    - Vazio - nulo;
- Sem preocupações com NullPointerException

Principais métodos:

Para criação do objeto:

> **Formato ->** Optional<{Type}> {ObjName }= Optional.{creationMedoth}();

- **of()** - Quando o valor não é nulo, caso seja nulo, joga uma exceção.
- **ofNullable()** - O valor atribuído pode ser nulo.
- **empty()** - sem parâmetros, objeto vazio.

Para acesso e manipulação:

- **ifPresentOrElse(func1, func2)** - se o valor do objeto for presente, faz a função 1, se não, a 2. As funções passadas geralmente são expressões lambdas.
- **ifPresent(func)/ifEmpty(func)** - o que fazer em cada caso, separadamente.
- **isEmpty()/isPresent()** - retorna um boolean.

Exemplo de código:

```
//Importando
import java.util.Optional;

public class Main {
    // Criando uma func que imprime na tela o parâmetro:
    static void print(String content) {
        System.out.println(content);
    }
    public static void main(String[] args) {
        // Cria um obj tipo Optional<String>
        Optional<String> optStr;
        // Atribui um valor:
        optStr = Optional.of("value kk");
        // Se o valor for presente, imprime o valor; se não imprime "tem nada
kkk"
        optStr.ifPresentOrElse((v) -> print(v), () -> print("tem nada kkk"));
        // Printa: value kk

        // Atribui um novo valor, dessa vez, nulo:
        optStr = Optional.ofNullable(null);
        optStr.ifPresentOrElse((v) -> print(v), () -> print("tem nada kkk"));
        // Printa: tem nada kkk

        // Tenta atribuir um valor nulo com .of:
        try {
            optStr = Optional.of(null);
            // (Throw: java.lang.NullPointerException)
        } catch (java.lang.NullPointerException e) {
            print("Ih ala, deu erro kkkkkkkk");
        }
    }
}
```

Streams - Dominando fluxo de dados

Aula única - auto-intitulada:

- Manipulação de coleções;
- Com paradigma funcional de forma paralela;
- Mais simples e performático;
- Imutabilidade - Não altera a coleção de origem, cria uma nova coleção e a retorna;
- Principais funcionalidades:
    - **Mapping** - retorna uma coleção de mesmo length, mas com os itens alterados em função de um algoritmo específico;

- **Filtering** - Filtra os elementos em função de um algoritmo específico e pode retornar uma lista maior ou igual à de origem;
- **ForEach** - Executa uma determinada lógica para cada elemento da coleção, não tem retorno;
- **Peek** - Faz o mesmo que ForEach, mas retorna a mesma coleção;
- **Counting** - Retorna o númro de elementos, em int;
- **Grouping** - Agrupa elementos em função de um determinado algoritmo;

Exemplo de código:

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<String> nomes = new ArrayList<>();

        nomes.add("Luis");
        nomes.add("Gustavo");
        nomes.add("Luiza");
        nomes.add("Rafael");
        nomes.add("Pedro");
        nomes.add("Felipe");
        nomes.add("Guilherme");
        nomes.add("Toshio");
        nomes.add("Vitoria");
        nomes.add("Bruno");

        System.out.println("> Qtd de itens: " + nomes.stream().count());
        System.out.println("> Mais letras: " +
nomes.stream().max(Comparator.comparingInt(String::length)));
        System.out.println("> Menos letras: " +
nomes.stream().min(Comparator.comparingInt(String::length)));
        System.out.println("> Não tem \"a\": " + nomes.stream().filter((nome) ->
                                                !
(nome.toUpperCase().contains("A"))).collect(Collectors.toList()));
        System.out.println("> Caixa alta e n de letras: " +
nomes.stream().map((nome) ->
                                                nome.toUpperCase().concat(" - "
+ String.valueOf(nome.length()))).collect(Collectors.toList()));
        System.out.println("> Tres primeiros: " +
nomes.stream().limit(3).collect(Collectors.toList()));

    }
}
```

Certificado \P/

https://certificates.digitalinnovation.one/843E5AFA