Universidade Federal de Ouro Preto

CSI032 - Programação de Computadores II

Collections: List

Professor: Dr. Rafael Frederico Alexandre
Contato: rfalexandre@decea.ufop.br
Colaboradores: Eduardo Matias Rodrigues
Contato: eduardo.matias@aluno.ufop.edu.br

Collections framework

Collections

Coleção
Visão geral

Coleção

Estrutura de dados que armazenam a referências de outros objetos.

1. ArrayList<String> nomes;
   1. Estrutura que armazena a referência de objetos do tipo String;
2. ArrayList<Computador> computadores;
   1. Armazena referência à instâncias da classe computadores.

Collections
Introdução

1. ArrayList é apenas uma coleção presente no Collections framework (presente no pacote java.util) do java. Conhecendo a hierarquia Collection:
   1. podemos programar de forma polimorfica (diminuir o acoplamento);
   2. podemos utilizar um conjunto pronto de algoritmos de alto desempenho, aumentando nossa produtividade.

Collections
Hierarquia

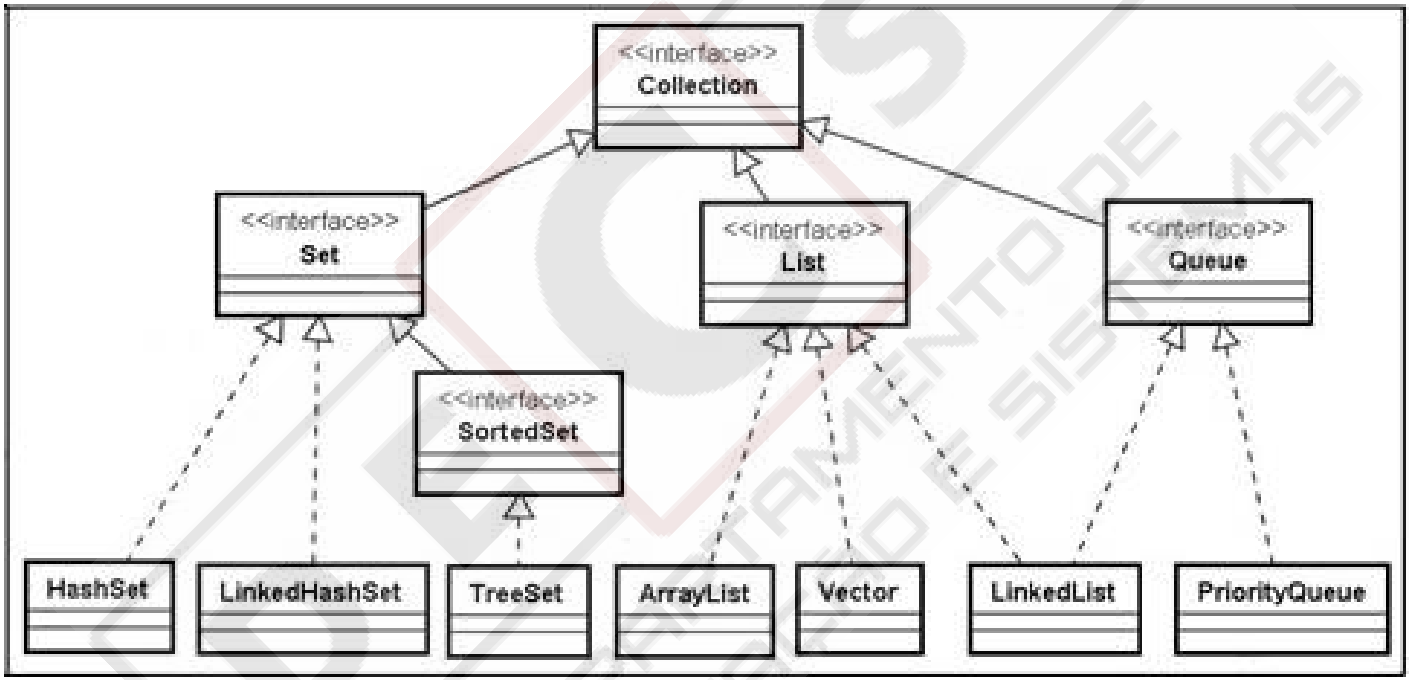


Figura: https://www.devmedia.com.br/java-collections-como-utilizar-collections/18450

Collections
Maps

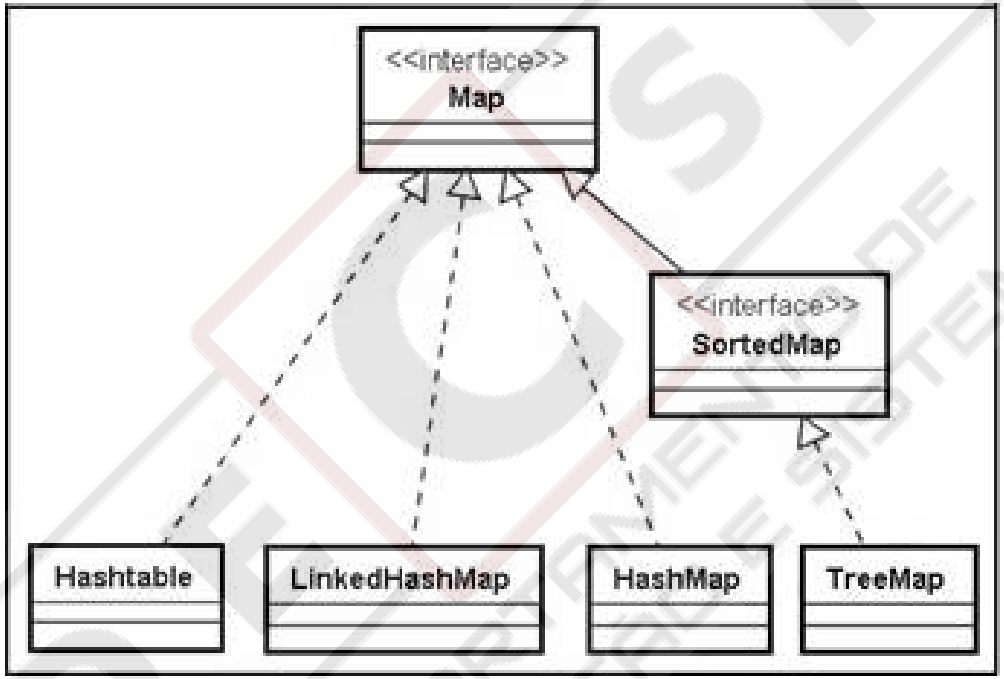


Figura: https://www.devmedia.com.br/java-collections-como-utilizar-collections/18450

1. Mesmo não derivando da interface collection, podemos considerá-las coleções.

Collections
Maps

1. A escolha da coleção a ser utilizada depende do tipo de problema envolvido;
2. Focaremos apenas na Interface **List** e na classe **ArrayList**;
3. O Collection Framework possui uma extensa documentação, que pode ser encontrada em:

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Interfaces
Descrição

1. **Collection**: está no topo da hierarquia, a partir da qual as interfaces *Set*, *Queue* e *List* são derivadas. Essa interface define as operações básicas que uma coleção deve possuir: inserir elemento, remover, pesquisar...
2. **List**: coleção ordenada (elementos ficam na ordem em que foram inseridos) que pode conter duplicatas;
3. **ArrayList**: implementa a Interface **List**, utiliza um Array estático de forma interna.

Collections

List
Hierarquia

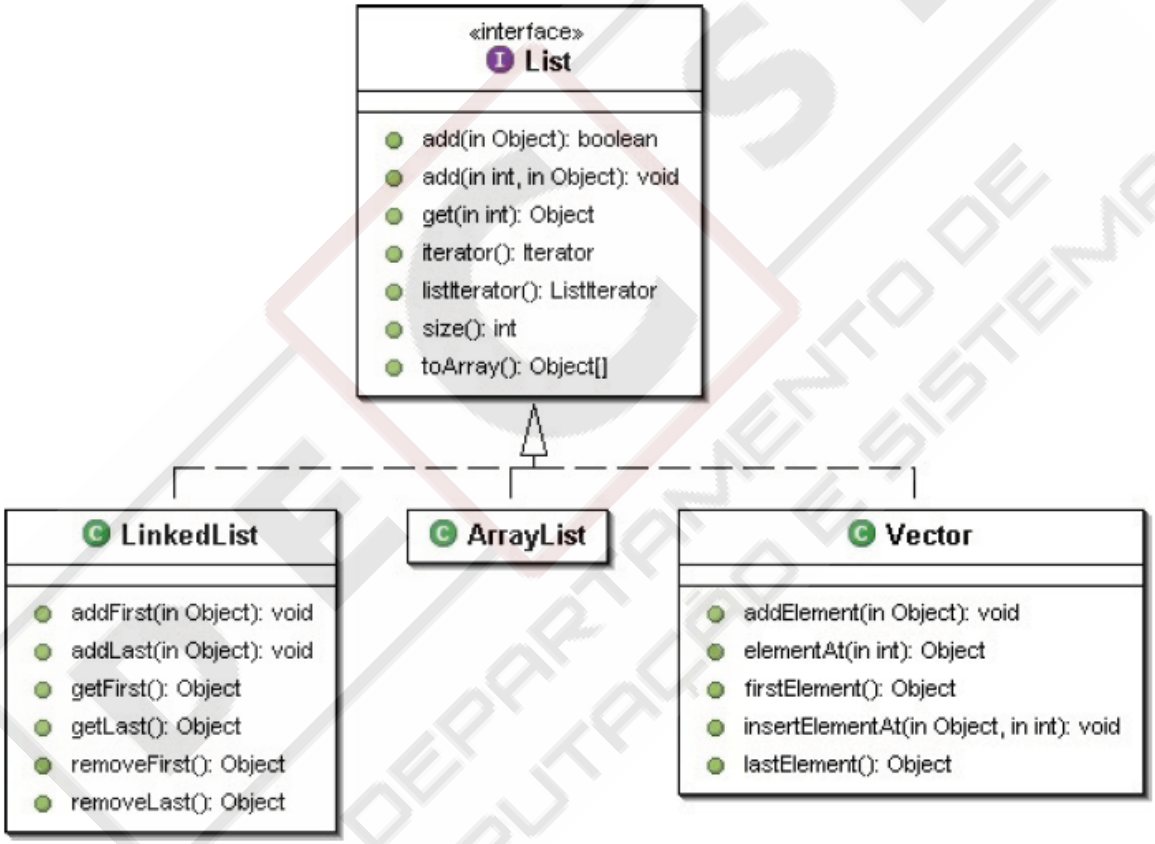


Figura: [Caelum, 2019]

List
LinkedList

1. **LinkedList**: ao contrário da classe ArrayList que utiliza vetores para armazenar os objetos, em uma LinkedList cada objeto contém a referência do próximo;
2. **Vector**: comportamento quase idêntico ao de um ArrayList, mas um Vector é Thread-safe, ou seja, permite acesso concorrente (várias threads compartilhando o mesmo Vector).

Collections

List
Polimorfismo

1. Sempre que pudermos, devemos programar da forma mais genérica possível (utilizando polimorfismo);
2. Tomemos um simples exemplo de um programa que armazena uma lista de nomes em um ArrayList e em seguida chama um método para exibir esses nomes;

List
Exemplo 1

```
public static void main(String[] args) {

    String[] vetNomes = {"eduardo", "alice", "gustavo"};
    ArrayList<String> nomes = new ArrayList<>();

    for (String nome : vetNomes) {
        nomes.add(nome);
    }

    exibirNomes(nomes);
}

public static void exibirNomes(ArrayList<String> nomes) {
    System.out.println("Nomes");
    for (String nome : nomes) {
        System.out.println(nome);
    }
}
```

List

Exemplo 1: problema

1. Se precisarmos chamar o método exibirNome() passando um outro tipo de **List** teremos um erro de tipo, já que especificamos que o método recebe apenas ArrayList's.

List

Exemplo 1, Cont.

```
public static void main(String[] args) {

    String[] vetNomes = {"eduardo", "alice", "gustavo"};
    LinkedList<String> nomes = new LinkedList<>();

    for (String nome : vetNomes) {
        nomes.add(nome);
    }

    exibirNomes(nomes);
}

public static void exibirNomes(ArrayList<String> nomes) {
    System.out.println("Nomes");
    for (String nome : nomes) {
        System.out.println(nome);
    }
}
```

List

Exemplo 1, Cont.

1. Se tentarmos compilar o programa teremos um erro de incompatibilidade de tipos:
    1. Exception in thread "main"java.lang.RuntimeException: Uncompilable source code - Erroneous sym type: colecoes.Colecoes.exibirNomes at colecoes.Colecoes.main(Colecoes.java:35)

List
Polimorfimo

1. Uma vez que **ArrayList**, **Vector** e **LinkedList** implementam a Interface **List** podemos referência-las como uma List!

```
List<String> nomes = new ArrayList<>(); // funciona
List<Integer> numeros = new LinkedList<>(); // funciona
List<Character> caracteres = new Vector<>(); // funciona
```

List

Polimorfismo, exemplo

```
public static void main(String[] args) {

    String[] vetNomes = {"eduardo", "alice", "gustavo"};

    List<String> nomesAL = new ArrayList<>(); // funciona
    List<String> nomesLL = new LinkedList<>(); // funciona
    List<String> nomesVet = new Vector<>(); // funciona

    for (String nome : vetNomes) {
        nomesAL.add(nome);
        nomesLL.add(nome);
        nomesVet.add(nome);
    }

    exibirNomes(nomesAL);
    exibirNomes(nomesLL);
    exibirNomes(nomesVet);
}

public static void exibirNomes(List<String> nomes) {
    System.out.print("Nomes: ");
    for (String nome : nomes) {
        System.out.print(nome + " | ");
    }
    System.out.println("");
}
```

List

Polimorfismo, exemplo (saída)

```
Nomes: eduardo | alice | gustavo |
Nomes: eduardo | alice | gustavo |
Nomes: eduardo | alice | gustavo |
```

List

Polimorfismo, exemplo (Cont.)

1. Ao declarar o parâmatro como uma **List<String>** permitimos que toda lista pode utilizar o método, sem este artifício teríamos que criar 3 métodos, um para cada tipo de lista: **ArrayList<String>**, **LinkedList<String>** e **Vector<String>**!
    1. Por isso, sempre opte por programar da forma mais genérica.
    2. O mesmo vale para o tipo de retorno dos métodos, opte sempre pelo retorno mais genérico, se possível.
2. As diferenças entre **ArrayList**, **LinkedList** e **Vector** são sentidas principalmente no desempenho em realizar determinadas operações. Consulte a documentação para mais detalhes;

Collections

Collections Framework
Operações

1. O Collections Framework nos trás um conjunto pronto de métodos de alto desempenho para realizar operações clássicas sob coleções;
    1. Esses métodos estão implementados na classe **Collections**.

Operações
Métodos

1. **sort**: ordena os elementos de uma lista;
2. **reverse**: inverte ordem dos elementos da lista;
3. **shuffle**: embaralha os elementos da **List**;
4. **min**: menor elemento da lista;
5. **max**: maior elemento da lista;
6. **frequency**: ocorrências de um elemento na coleção;
7. **disjoint**: verifica se duas coleções não tem nenhum elemento em comum;
8. Todos estes métodos são estáticos.

Sort
Exemplo

```
public static void main(String[] args) {

    Integer[] numVet = {10, 23, -1, -10, 4353, 231, -90, 0, 1};
    List<Integer> numeros = new ArrayList<>();

    for (Integer num : numVet) {
        numeros.add(num);
    }

    System.out.println("Ordem de inserção: " + numeros);
    Collections.sort(numeros); // Altera a lista original
    System.out.println("Ordem crescente: " + numeros);
}
```

Sort
saída do programa

```
Ordem de inserção: [10, 23, -1, -10, 4353, 231, -90, 0, 1]
Ordem crescente: [-90, -10, -1, 0, 1, 10, 23, 231, 4353]
```

Sort
Cont.

1. E se quisessemos ordenar uma lista de objetos criados por nós? Exemplo:
2. Suponha que uma loja de informática tenha armazenado em seu sistema uma lista de todos os computadores da loja. Para cada computador sabemos seu modelo, preço e data de fabricação e queremos ordenar as máquinas do mais barato para o mais caro, como fazer isso?
3. Se tentarmos chamar o método sort como fizemos para uma lista de inteiros vamos ter um erro;

Sort
Cont.

```
public class Computador {

    private String modelo;
    private String dataFabricacao;
    private double preco;

    public Computador(String modelo, String dataFabricacao,
            double preco) {
        this.modelo = modelo;
        this.dataFabricacao = dataFabricacao;
        this.preco = preco;
    }
}
```

Sort
Cont.

```
public static void main(String[] args) {

    List<Computador> computadores = new ArrayList<>();

    computadores.add(new Computador("Samsung Expert", "2015", 3000));
    computadores.add(new Computador("Dell Inspiron", "2015", 2000));
    computadores.add(new Computador("Lenovo ThinkPad", "2019", 3500));

    Collections.sort(computadores);

}
```

Sort
Cont.

1. Esse error acontece porque o método sort() espera uma lista que implementa a interface **Comparable**;
2. As classes empacotadoras do java: Double, Integer, Float, Character, Boolean, Long, Short e Byte implementam a interface Comparable, por isso conseguimos ordenar nossa lista de inteiros (e qualquer outra lista de objetos do tipo de uma Wrapper Class) sem problemas;

Sort
Comparable

1. A Interface Comparable possui apenas um método, o compareTo();
   1. Deve retornar 0 se o objeto comparado for **iqual** a este objeto;
   2. Deve retornar -1 se este objeto for **menor** que o objeto comparado;
   3. Deve retorn 1 se este objeto for **maior** que o objeto comparado
2. O método sort da classe Collections usa o método compareTo como critério de ordenação.

Sort
Classe Computador

```
public class Computador implements Comparable<Computador> {

    private String modelo;
    private String dataFabricacao;
    private double preco;

    public Computador(String modelo, String dataFabricacao,
            double preco) {...5 lines }


    @Override
    public int compareTo(Computador t) {
        if (this.preco < t.preco)
            return -1;
        if (this.preco > t.preco)
            return 1;
        return 0;
    }
```

Sort
Principal

```
public static void main(String[] args) {
    List<Computador> computadores = new ArrayList<>();

    computadores.add(new Computador("Samsung Expert", "2015", 3000));
    computadores.add(new Computador("Dell Inspiron", "2015", 2000));
    computadores.add(new Computador("Lenovo ThinkPad", "2019", 3500));

    System.out.println("Antes de ordenar: \n");
    print(computadores);

    Collections.sort(computadores);

    System.out.println("Depois de ordenar: \n");
    print(computadores);
}

public static void print(List<Computador> computadores) {
    for (Computador c : computadores) {
        System.out.println("Modelo: " + c.getModelo());
        System.out.println("Fabricação: " + c.getDataFabricacao());
        System.out.println("Preço: " + c.getPreco());
        System.out.println("");
    }
}
```

Sort

Saída do programa

```
Antes de ordenar:

Modelo: Samsung Expert
Fabricação: 2015
Preço: 3000.0

Modelo: Dell Inspiron
Fabricação: 2015
Preço: 2000.0

Modelo: Lenovo ThinkPad
Fabricação: 2019
Preço: 3500.0
```

Sort

Saída do programa

```
Depois de ordenar:

Modelo: Dell Inspiron
Fabricação: 2015
Preço: 2000.0

Modelo: Samsung Expert
Fabricação: 2015
Preço: 3000.0

Modelo: Lenovo ThinkPad
Fabricação: 2019
Preço: 3500.0
```

Sort
Interface Comparator

1. A interface Comparable só nos permite ordenar uma lista utilizando um único atributo como critério de ordenação, mas se quisermos ordenar nossos computadores pela data de fabricação?
    1. Poderíamos alterar o método *compareTo()* para que comparasse as datas, o que é uma má prática de programação, já que precisariamos alterar esse método cada vez que quissésmos trocar o atributo que será comparado;
    2. O método *sort()* assim como outros métodos da classe Collections que necessita comparar objetos, são sobrecarregados e possuem uma segunda implementação que além da lista recebe um **Comparator**.

Sort
Interface Comparator

1. Além da possibilidade de ordenar os computadores pelo preço, queremos ordená-los também pela data de fabricação (tipo da data foi alterado de **String** para **LocalDate**);
2. Para isso temos que definir como as datas serão comparadas.

Interface Comparator

Comparando atributos

1. Para cada novo atributo que você deseja ser usado como critério de comparação:
   1. Crie uma nova classe e faça esta classe implementar a interface **Comparator<Classe>**, onde **Classe** é a classe do atributo que será comparado;
   2. Sobrescreva o método *compare()* (retorna -1, 0 ou 1, com os mesmos critérios do método *compareTo()* da interface **Comparable**)

Sort
Comparador de datas

```
public class DataComparator implements Comparator<Computador> {

    @Override
    public int compare(Computador c1, Computador c2) {
        if (c1.getDataFabricação().isBefore(c2.getDataFabricação()))
            return -1;
        if (c1.getDataFabricação().isAfter(c2.getDataFabricação()))
            return 1;
        return 0;
    }
}
```

Sort
Classe de teste

```
public static void main(String[] args) {
    List<Computador> computadores = new ArrayList<>();
    DataComparator dataC = new DataComparator();

    // Modelo, Preço, dia, mes e ano de fabricação
    Computador c1 = new Computador("Dell Inspiron", 2000, 14, 5, 2015);
    Computador c2 = new Computador("Samsun Expert", 2500, 10, 10, 2018);
    Computador c3 = new Computador("Lenovo", 2000, 20, 5, 2015);

    computadores.add(c1);
    computadores.add(c2);
    computadores.add(c3);

    // Recebe também um comparador de datas
    Collections.sort(computadores, dataC);

    for (Computador c : computadores) {
        System.out.println("Modelo: " + c.getModelo());
        System.out.println("Fabricação: " + c.getDataFabricação());
        System.out.println("Preço: " + c.getPreco() + "\n");
    }
```

Sort

Saída do programa

```
Modelo: Dell Inspiron
Fabricação: 2015-05-14
Preço: 2000.0

Modelo: Lenovo
Fabricação: 2015-05-20
Preço: 2000.0

Modelo: Samsun Expert
Fabricação: 2018-10-10
Preço: 2500.0
```

Ordem reversa

reverse

```
public static void main(String[] args) {
    Integer[] numVet = {12, 123, -2, -1, 0, 21, 1335, -44};
    List<Integer> numeros = new ArrayList<>();

    for (Integer i : numVet) {
        numeros.add(i);
    }

    System.out.println("Ordem de inserção: " + numeros);
    Collections.reverse(numeros);
    System.out.println("Ordem reversa: " + numeros);
}
```

```
Ordem de inserção: [12, 123, -2, -1, 0, 21, 1335, -44]
Ordem reversa: [-44, 1335, 21, 0, -1, -2, 123, 12]
```

Embaralhar
shuffle

```
public static void main(String[] args) {
    Integer[] numVet = {12, 123, -2, -1, 0, 21, 1335, -44};
    List<Integer> numeros = new ArrayList<>();

    for (Integer i : numVet) {
        numeros.add(i);
    }

    System.out.println("Ordem de inserção: " + numeros);
    Collections.shuffle(numeros);
    System.out.println("Embaralhado: " + numeros);
}
```

```
Ordem de inserção: [12, 123, -2, -1, 0, 21, 1335, -44]
Embaralhado: [123, -2, -1, -44, 21, 0, 1335, 12]
```

Min e Max

min e max

```
public static void main(String[] args) {
    Integer[] numVet = {12, 123, -2, -1, 0, 21, 1335, -44};
    List<Integer> numeros = new ArrayList<>();

    for (Integer i : numVet) {
        numeros.add(i);
    }

    int min = Collections.min(numeros);
    int max = Collections.max(numeros);
    System.out.println("Min: " + min + " | Max: " + max);
}
```

```
Min: -44 | Max: 1335
```

Min e Max

Data

1. Os métodos **min** e **max** também são sobrecarregados e possuem uma segunda implementação que recebe um Comparator;
2. Se quisermos saber qual é o computador com a data de fabricação mais recente e a mais antiga, basta passar nosso comparador de data para esses método;

Min e Max

Teste

```
public static void main(String[] args) {
    List<Computador> computadores = new ArrayList<>();
    DataComparator dataC = new DataComparator();

    // Modelo, Preço, dia, mes e ano de fabricação
    Computador c1 = new Computador("Dell Inspiron", 2000, 14, 5, 2015);
    Computador c2 = new Computador("Samsun Expert", 2500, 10, 10, 2018);
    Computador c3 = new Computador("Lenovo", 2000, 20, 5, 2015);

    computadores.add(c1);
    computadores.add(c2);
    computadores.add(c3);

    Computador maisAntigo = Collections.min(computadores, dataC);
    Computador maisNovo = Collections.max(computadores, dataC);

    System.out.println("Mais antigo");
    maisAntigo.exibirDados();
    System.out.println("Mais recente");
    maisNovo.exibirDados();
```

Min e Max

Saída

```
Mais antigo
Modelo: Dell Inspiron
Fabricação: 2015-05-14
Preço: 2000.0

Mais recente
Modelo: Samsun Expert
Fabricação: 2018-10-10
Preço: 2500.0
```

Frequência
frequency

```
public static void main(String[] args) {
    String[] vet = {"eduardo", "marias", "fernanda", "joao", "eduardo", "ma
    List<String> pessoas = new ArrayList<>();

    for (String nome : vet) {
        pessoas.add(nome);
    }

    int n = Collections.frequency(pessoas, "eduardo");
    System.out.println("eduardo aparece " + n + " vezes");
}
```

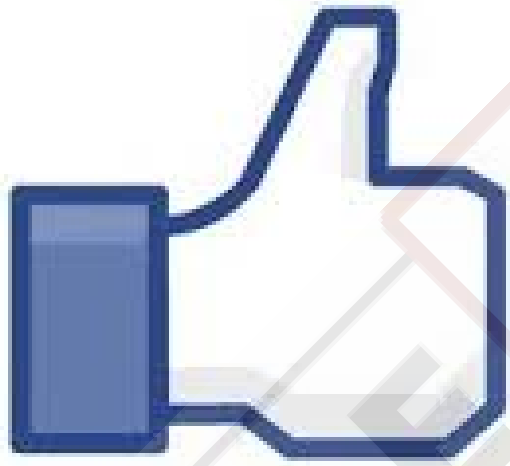
```
eduardo aparece 2 vezes
```

Disjunta
disjoint

```
public static void main(String[] args) {
    String[] vet = {"eduardo", "maria", "fernanda", "joao", "marcelo"};
    String[] vet2 = {"eduardo", "fernanda", "marcelo"};

    List<String> pessoas = new ArrayList<>();
    List<String> pessoas2 = new ArrayList<>();

    for (String nome : vet) {
        pessoas.add(nome);
    }

    for (String nome: vet2) {
        pessoas2.add(nome);
    }

    if (Collections.disjoint(pessoas, pessoas2)) {
        System.out.println("Listas não tem pessoas em comum");
    } else {
        System.out.println("Listas tem pessoas em comum");
    }
}
```

Disjunta
Cont. Saída

```
Listas tem pessoas em comum
```



MUITO OBRIGADO!!!

Referências Bibliográficas I

Caelum (2019).
Java e orientação a objeto.
https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf.