# Linguagem de Programação I

## Aula 6
## Coleções de Objetos Genéricas
## Classes e Métodos

l.bertholdo@ifsp.edu.br

# Conteúdo

- Coleções de Objetos
- Listas de Objetos
  - ArrayList e LinkedList
- Conjuntos de Objetos
  - HashSet e TreeSet
- Mapas de Objetos
  - HashMap e TreeMap

*l.bertholdo@ifsp.edu.br*

# Coleções de Objetos

*As coleções de objetos são implementadas por meio de **classes** e **interfaces** presentes no pacote **java.util** da API Java, o qual deve ser importado pelas classes que as utilizarem.*

- *Arrays* são muito úteis, porém têm algumas limitações:
  - Não é possível aumentar ou diminuir seu tamanho.
  - A ordenação dos elementos de um *array* não é automática.
  - Seus índices precisam ser sempre números inteiros para que os dados possam ser acessados.
- As coleções visam facilitar o uso de estruturas de dados como listas, filas, pilhas, conjuntos e outras, tornando a manipulação destas estruturas muito mais simples e intuitiva.
- Uma das principais vantagens é que, ao criar uma coleção, seu tamanho não precisa ser especificado, já que ele aumenta ou diminui conforme os elementos são inseridos ou removidos.

*l.bertholdo@ifsp.edu.br*

# Coleções de Objetos

*A classe **Object** é mãe, direta ou indiretamente, de todas as classes do Java, sejam classes da API Java ou classes criadas pelo desenvolvedor. Assim, quando se declara uma nova classe esta classe é automaticamente filha de **Object**.*

- As antigas coleções da API Java não eram **genéricas**, pois conseguiam armazenar apenas instâncias da classe **Object**.
- Com isso, ao armazenar um elemento na coleção, essas classes precisavam converter o tipo de dado do elemento para o tipo **Object** e, ao recuperá-los, precisam fazer a operação inversa.
- As atuais coleções não estão vinculadas ao tipo **Object**, o que as torna capazes de armazenar qualquer tipo de dado. Por isso, são consideradas "coleções genéricas".

```
ArrayList lista1 = new ArrayList(); // Armaneza objetos do tipo Object.
ArrayList<String> lista2 = new ArrayList<>(); // Armaneza objetos do tipo String.
ArrayList<Integer> lista3 = new ArrayList<>(); // Armaneza objetos do tipo Integer.
ArrayList<Produto> lista4 = new ArrayList<>(); // Armaneza objetos do tipo Produto.
```

# Coleções de Objetos

- As classes da API Java que representam coleções armazenam somente instâncias de classes.

- Assim, para que uma coleção possa armazenar um tipo de dado nativo (int, float, double, char etc), é preciso associá-la à classe da API Java que equivale a este tipo de dado.

```
ArrayList<Integer> lista1 = new ArrayList<>(); // Armaneza objetos do tipo Integer.
ArrayList<int> lista2 = new ArrayList<>(); // -> Erro, pois "int" é um tipo nativo.
```

*l.bertholdo@ifsp.edu.br*

# Coleções de Objetos

- Cada tipo nativo tem sua respectiva classe, capaz de converter e manipular valores do seu tipo correspondente.

| Tipo Nativo | Classe Equivalente |
|---|---|
| int | Integer |
| float | Float |
| double | Double |
| long | Long |
| char | Character |
| boolean | Boolean |

- A conversão entre **tipos nativos** e **classes** é possível graças aos mecanismos ***autoboxing*** *e* ***autounboxing*** da linguagem Java:

```
Integer x = 2; // Autoboxing (conversão de int para Integer)
int y = x;     // Autounboxing (conversão de Integer para int)
x = y;         // Autoboxing (conversão de int para Integer)
```

# Coleções de Objetos

- Uma boa prática é declarar a coleção com o tipo da interface que ela implementa, e definir a classe de implementação ao instanciá-la com a palavra **new**.

**List** é uma interface que define o conjunto de métodos das **coleções de listas** da API Java. **ArrayList** é uma das implementações desta interface*.

```
List<Integer> lista = new ArrayList<>();
```

Assim, se for preciso alterar a classe de implementação, bastará instanciar esta nova classe.

```
lista = new LinkedList<>();
```

* Classes que implementam a interface **List**:
https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/List.html

*l.bertholdo@ifsp.edu.br*

# Listas de Objetos

- Listas de objetos funcionam de forma similiar a um *array* comum, porém com recursos adicionais e a diferença de que a alocação dos elementos na memória é feita dinamicamente.

- As listas de objetos são definidas por classes que implementam a interface **List**, a qual declara métodos específicos para manipulação de listas.

- Nas listas de objetos, os elementos podem se repetir.

*l.bertholdo@ifsp.edu.br*

# Listas de Objetos

- Algumas das classes que implementam a interface **List** visando a representação de listas:
  - **ArrayList** – Possui um mecanismo mais rápido que um *array* comum, exceto para as operações de inserção e remoção.
  - **LinkedList** – Possui um mecanismo mais rápido para operações de inserção e remoção, porém é mais lento para acesso sequencial aos elementos da lista.
- Ao criar uma instância destas classes, é preciso definir o tipo de objeto que a lista irá armazenar. Por exemplo:
  List<**Integer**> lista = new ArrayList<>();

# ArrayList e LinkedList

- Principais métodos:
  - **add:** Adiciona um elemento no final ou em uma dada posição da lista, considerando que a 1ª posição é zero.

    **Sintaxe:** <nome_lista>.add(<elemento>)
    **Sintaxe:** <nome_lista>.add(<posição>, <elemento>)
  - **remove:** Remove um elemento da lista por meio de sua posição ou por meio da 1ª ocorrência do elemento na lista.

    **Sintaxe:** <nome_lista>.remove(<posição*> ou <elemento>)
  - **get:** Retorna um elemento por meio de sua posição na lista.

    **Sintaxe:** <nome_lista>.get(<posição>)
  - **size:** Retorna quantos elementos a lista possui.

    **Sintaxe:** <nome_lista>.size()
  - **clear:** Remove todos os elementos da lista.

    **Sintaxe:** <nome_lista>.clear()

* Quando a lista armazena o tipo **Integer**, para evitar ambiguidade, o parâmetro informado indica a posição, e não o elemento.

*l.bertholdo@ifsp.edu.br*

# ArrayList e LinkedList

```java
public static void main(String[] args) {
    // Cria uma lista de objetos ArrayList vazia.
    List<String> alunos = new ArrayList<>();

    //Adiciona os elementos à lista.
    alunos.add("João");
    alunos.add("Maria");
    alunos.add("Pedro");
    alunos.add("Carlos");
    // Adicionando o elemento "João" novamente, agora na posição 2.
    alunos.add(2, "João");
    System.out.println(alunos); // Imprime os elementos da lista.

    alunos.remove("Pedro"); // Remove o elemento "Pedro" da lista.
    alunos.remove(0); // Remove o elemento que está na posição zero na lista.
    System.out.println(alunos); // Imprime os elementos da lista.

    // Obtém o elemento que está na posição 1 na lista.
    System.out.println("2º aluno: " + alunos.get(1));

    // Retorna o número de elementos contidos na lista.
    System.out.println("Número de alunos: " + alunos.size());
}
```

```
[João, Maria, João, Pedro, Carlos]
[Maria, João, Carlos]
2º aluno: João
Número de alunos: 3
```

**ArrayLists** e **LinkedLists** também podem ser inicializados já em sua declaração por meio da classe **Arrays**:

```java
List<String> alunos = new ArrayList<>(Arrays.asList("João", "Maria", "Pedro"));
```

# ArrayList e LinkedList

- Principais métodos:
  - **set:** Substitui um elemento em uma dada posição da lista, considerando que a 1ª posição é zero.
    **Sintaxe:** <nome_lista>.set(<posição>, <novo_elemento>)
  - **contains:** Retorna se uma lista contém um determinado elemento.
    **Sintaxe:** <nome_lista>.contains(<elemento>)
  - **indexOf:** Retorna a posição da 1ª ocorrência do elemento. Se o elemento não existir, retorna -1.
    **Sintaxe:** <nome_lista>.indexOf(<elemento>)
  - **lastIndexOf:** Retorna a posição da última ocorrência do elemento. Se o elemento não existir, retorna -1.
    **Sintaxe:** <nome_lista>.lastIndexOf(<elemento>)
  - **isEmpty:** Retorna se uma lista está vazia.
    **Sintaxe:** <nome_lista>.isEmpty()

# ArrayList e LinkedList

```
[João, Maria, Carlos, Julia, Carlos, Maria]
Sim, a lista contém a aluna Julia.
Posição da 1ª ocorrência de 'Maria': 1
Posição da última ocorrência de 'Carlos': 4
A lista não está vazia.
```

```java
public static void main(String[] args) {
    // Cria uma lista de objetos ArrayList.
    List<String> alunos = new ArrayList<>
                          (Arrays.asList("João", "Maria", "Carlos", "Pedro", "Carlos", "Maria"));

    alunos.set(3, "Julia"); // Substitui o elemento "Pedro" por "Julia".
    System.out.println(alunos); // Imprime os elementos da lista.

    if (alunos.contains("Julia")) // Verifica se a lista contém o elemento "Julia".
        System.out.println("Sim, a lista contém a aluna Julia.");      "Julia"

    // Retorna a posição da 1ª ocorrência do elemento "Maria".
    System.out.println("Posição da 1ª ocorrência de 'Maria': " + alunos.indexOf("Maria"));

    // Retorna a posição da última ocorrência do elemento "Carlos".
    System.out.println("Posição da última ocorrência de 'Carlos': " + alunos.lastIndexOf("Carlos"));

    if (!alunos.isEmpty()) // Verifica se a lista está vazia.
        System.out.println("A lista não está vazia.");
}
```

l.bertholdo@ifsp.edu.br

# LinkedList – Filas e Pilhas

- A classe **LinkedList** possui métodos adicionais que permitem simular estruturas de dados como filas e pilhas.
  - **Fila** – Estrutura de dados onde o 1º elemento a entrar é o 1º a sair (*FIFO – first-in, first-out*). É simulada por meio de métodos para inserção no fim (**add** ou **addLast**) e remoção no início (**poolFirst**) de uma lista.
  - **Pilha** – Estrutura de dados onde o último elemento a entrar é o 1º a sair (*LIFO – last-in, first-out*). É simulada por meio de métodos para inserção no fim (**add** ou **addLast**) e remoção no fim (**poolLast**) de uma lista.

# LinkedList – Filas e Pilhas

- Principais métodos:
  - **addFirst:** Insere um elemento na 1ª posição de uma lista.
    **Sintaxe:** <nome_lista>.addFirst(<elemento>)
  - **addLast:** Insere um elemento na última posição de uma lista.
    **Sintaxe:** <nome_lista>.addLast(<elemento>)
  - **pollFirst:** Remove o elemento que está na 1ª posição de uma lista.
    **Sintaxe:** <nome_lista>.pollFirst()
  - **pollLast:** Remove o elemento que está na última posição de uma lista.
    **Sintaxe:** <nome_lista>.pollLast()
  - **peekFirst:** Retorna o elemento que está na 1ª posição de uma lista.
    **Sintaxe:** <nome_lista>.peekFirst()
  - **peekLast:** Retorna o elemento que está na última posição de uma lista.
    **Sintaxe:** <nome_lista>.peekLast()

# LinkedList – Fila

Dependendo dos métodos a serem usados, é necessário usar outra interface implementada pela classe de coleção. Por exemplo, a interface **List** não possui os métodos **addLast**, **peekFirst** e **pollFirst**. Logo, não será possível declarar a coleção com este tipo de interface (ocorrerá erro de compilação ao chamar estes métodos). Nesse caso, pode-se usar a interface **Deque**, que também é implementada pela classe **LinkedList** e possui estes três métodos.

```
public static void main(String[] args) {
    // Cria uma fila de objetos LinkedList vazia.
    Deque<String> fila = new LinkedList<>();

    //Adiciona os elementos à fila.
    fila.add("João");
    fila.add("Maria");
    fila.add("Pedro");
    fila.add("Carlos");

    // Adiciona o elemento "Julia" na última posição da fila.
    fila.addLast("Julia");
    System.out.println(fila); // Imprime os elementos da fila.

    // Retorna o 1º elemento da fila.
    System.out.println("1º elemento da fila: " + fila.peekFirst());

    fila.pollFirst(); // Remove o 1º elemento da fila.
    System.out.println(fila); // Imprime os elementos da fila.
}
```

```
[João, Maria, Pedro, Carlos, Julia]
1º elemento da fila: João
[Maria, Pedro, Carlos, Julia]
```

l.bertholdo@ifsp.edu.br

# LinkedList

*É comum algumas classes implementarem mais de uma interface. Nesse caso, a **interface** associada à instância determinará quais métodos essa instância poderá utilizar.*

Métodos abstratos da interface **List**:
- Método a
- Método b

Métodos abstratos da interface **Deque**:
- Método a
- Método b
- Método c
- Método d

Métodos concretos implementados na classe **LinkedList**:
- Método a -> O método é implementado para **List** e **Deque**.
- Método b -> O método é implementado para **List** e **Deque**.
- Método c -> O método é implementado para **Deque**.
- Método d -> O método é implementado para **Deque**.

**List<String> fila = new LinkedList<>();**
Métodos acessíveis pela instância **fila**:
- fila.a
- fila.b

**Deque<String> fila = new LinkedList<>();**
Métodos acessíveis pela instância **fila**:
- fila.a
- fila.b
- fila.c
- fila.d

r.bertholdo@ifsp.edu.br

# LinkedList – Pilha

A interface **List** não possui os métodos **addLast**, **peekLast** e **pollLast**. Por isso, é usada novamente a interface **Deque**, que possui estes três métodos.

```
public static void main(String[] args) {
    // Cria uma pilha de objetos LinkedList.
    Deque<String> pilha = new LinkedList<>
                          (Arrays.asList("João", "Maria", "Pedro", "Carlos"));

    // Adiciona o elemento "Julia" no topo da pilha (última posição).
    pilha.addLast("Julia");
    System.out.println(pilha); // Imprime os elementos da pilha.

    // Retorna o elemento que está no topo da pilha (último elemento).
    System.out.println("Elemento que está no topo da pilha: " + pilha.peekLast());

    pilha.pollLast(); // Remove o elemento que está no topo da pilha (último elemento).
    System.out.println(pilha); // Imprime os elementos da pilha.
}
```

```
[João, Maria, Pedro, Carlos, Julia]
Elemento que está no topo da pilha: Julia
[João, Maria, Pedro, Carlos]
```

l.bertholdo@ifsp.edu.br

# Conjuntos de Objetos

- Os conjuntos de objetos são definidos por classes que implementam a interface **Set**, a qual declara métodos específicos para manipulação de conjuntos.
- Em um conjunto de objetos, não são permitidos objetos duplicados.

# Conjuntos de Objetos

- Algumas das classes que implementam a interface **Set** visando a representação de conjuntos:
  - **HashSet** – Possui um mecanismo mais rápido para operações de alteração em conjuntos, porém não estabelece uma ordem entre seus elementos.
  - **TreeSet** – Possui um mecanismo mais lento para operações de inserção e remoção em conjuntos, porém mantém a ordenação crescente de seus elementos.
- Ao criar uma instância destas classes, é preciso definir o tipo de objeto que o conjunto irá armazenar. Por exemplo:
  `Set<Integer> conjunto = new HashSet<>();`

# HashSet e TreeSet

- Principais métodos:
  - **add:** Adiciona um elemento ao conjunto.
    **Sintaxe:** <nome_conjunto>.add(<elemento>)
  - **remove:** Remove um elemento do conjunto.
    **Sintaxe:** <nome_conjunto>.remove(<elemento>)
  - **isEmpty:** Retorna se um conjunto está vazio.
    **Sintaxe:** <nome_conjunto>.isEmpty()
  - **contains:** Retorna se um conjunto contém um determinado elemento.
    **Sintaxe:** <nome_conjunto>.contains(<elemento>)
  - **size:** Retorna quantos elementos o conjunto possui.
    **Sintaxe:** <nome_conjunto>.size()
  - **clear:** Remove todos os elementos do conjunto.
    **Sintaxe:** <nome_conjunto>.clear()

*l.bertholdo@ifsp.edu.br*

# HashSet

Os nomes não são armazenados em ordem alfabética no conjunto.

```
public static void main(String[] args) {
    // Cria um conjunto de objetos HashSet vazio.
    Set<String> alunos = new HashSet<>();

    //Adiciona os elementos ao conjunto.
    alunos.add("João");
    alunos.add("Maria");
    alunos.add("Pedro");
    alunos.add("João"); // Tentando adicionar o elemento "João" novamente.
    alunos.add("Carlos");
    System.out.println(alunos); // Imprime os elementos do conjunto.

    alunos.remove("Pedro"); // Remove o elemento "Pedro" do conjunto.

    if (!alunos.isEmpty()) // Verifica se o conjunto está vazio.
        System.out.println("O conjunto não está vazio.");

    if (alunos.contains("Maria")) // Verifica se o conjunto contém o elemento "Maria".
        System.out.println("O conjunto contém a aluna Maria.");

    // Retorna o número de elementos contidos no conjunto.
    System.out.println("Número de alunos: " + alunos.size());

    System.out.println(alunos); // Imprime os elementos do conjunto.
}
```

```
[João, Pedro, Maria, Carlos]
O conjunto não está vazio.
O conjunto contém a aluna Maria.
Número de alunos: 3
[João, Maria, Carlos]
```

**HashSets** e **TreeSets** também podem ser inicializados já em sua declaração por meio da classe **Arrays**:

```
Set<String> alunos = new HashSet<>(Arrays.asList("João", "Maria", "Pedro"));
```

# TreeSet

Os nomes são armazenados em ordem alfabética no conjunto.

```
[Carlos, João, Maria, Pedro]
O conjunto não está vazio.
O conjunto contém a aluna Maria.
Número de alunos: 3
[Carlos, João, Maria]
```

```java
public static void main(String[] args) {
    // Cria um conjunto de objetos TreeSet vazio.
    Set<String> alunos = new TreeSet<>();

    //Adiciona os elementos ao conjunto.
    alunos.add("João");
    alunos.add("Maria");
    alunos.add("Pedro");
    alunos.add("João"); // Tentando adicionar o elemento "João" novamente.
    alunos.add("Carlos");
    System.out.println(alunos); // Imprime os elementos do conjunto.

    alunos.remove("Pedro"); // Remove o elemento "Pedro" do conjunto.

    if (!alunos.isEmpty()) // Verifica se o conjunto está vazio.
        System.out.println("O conjunto não está vazio.");

    if (alunos.contains("Maria")) // Verifica se o conjunto contém o elemento "Maria".
        System.out.println("O conjunto contém a aluna Maria.");

    // Retorna o número de elementos contidos no conjunto.
    System.out.println("Número de alunos: " + alunos.size());

    System.out.println(alunos); // Imprime os elementos do conjunto.
}
```

l.bertholdo@ifsp.edu.br

# TreeSet

- Outros métodos:
  - **first:** Retorna o 1º elemento do conjunto.
    **Sintaxe:** <nome_conjunto>.first()
  - **last:** Retorna o último elemento do conjunto.
    **Sintaxe:** <nome_conjunto>.last()
  - **headSet:** Retorna os elementos antecessores de um dado elemento. **Sintaxe:** <nome_conjunto>.headSet(<elemento>)
  - **tailSet:** Retorna um dado elemento e seus elementos sucessores. **Sintaxe:** <nome_conjunto>.tailSet(<elemento>)
  - **subSet:** Retorna um subconjunto que vai do 1º elemento informado até o antecessor do 2º elemento informado.
    **Sintaxe:** <nome_conjunto>.subSet(<1º elemento>, <2º elemento>)

# TreeSet

A interface **Set** não possui os métodos **first**, **last**, **headSet**, **tailSet** e **subSet**. Nesse caso, pode-se usar a interface **SortedSet**, que também é implementada pela classe **TreeSet** e possui estes métodos.

```
[Carlos, João, Maria, Pedro, Raquel]
1º aluno do conjunto: Carlos
Último aluno do conjunto: Raquel
Alunos antecessores de Maria: [Carlos, João]
Maria e seus alunos sucessores: [Maria, Pedro, Raquel]
Subconjunto: [João, Maria]
```

```
public static void main(String[] args) {
    // Cria um conjunto de objetos TreeSet.
    SortedSet<String> alunos = new TreeSet<>(Arrays.asList("João", "Maria",
                                                "Pedro", "Carlos", "Raquel"));
    System.out.println(alunos); // Imprime os elementos do conjunto.

    // Imprime o 1º elemento do conjunto.
    System.out.println("1º aluno do conjunto: " + alunos.first());

    // Imprime o último elemento do conjunto.
    System.out.println("Último aluno do conjunto: " + alunos.last());

    // Imprime os elementos antecessores do elemento "Maria".
    System.out.println("Alunos antecessores de Maria: " + alunos.headSet("Maria"));

    // Imprime o elemento "Maria" e seus sucessores.
    System.out.println("Maria e seus alunos sucessores: " + alunos.tailSet("Maria"));

    // Imprime o subconjunto que vai do elemento "João" até o antecessor de "Pedro".
    System.out.println("Subconjunto: " + alunos.subSet("João", "Pedro"));
}
```

# HashSet e TreeSet

- As classes **HashSet** e **TreeSet** também têm métodos para realizar operações matemáticas, como: **união**, **intersecção** e **diferença** entre conjuntos.
  - **addAll (união)** – Junta os elementos de dois conjuntos, descartando os elementos repetidos. O conjunto que chama o método é alterado passando a conter os elementos dos dois conjuntos.

    **Sintaxe:** <nome_conjunto1>.addAll(<nome_conjunto2>)
  - **retainAll (intersecção)** – Retorna os elementos que dois conjuntos têm em comum. O conjunto que chama o método é alterado, passando a conter apenas os elementos em comum.

    **Sintaxe:** <nome_conjunto1>.retainAll(<nome_conjunto2>)

# HashSet e TreeSet

- **removeAll (diferença entre conjuntos)** – Retorna os elementos de um conjunto que não estão no outro conjunto. O conjunto que chama o método é alterado passando a conter apenas os elementos que não estão no outro conjunto.

  **Sintaxe:** <nome_conjunto1>.removeAll(<nome_conjunto2>)

- **containsAll:** Este método não modifica o conjunto que o chama. Ele apenas retorna se o conjunto passado como argumento está contido no conjunto que chamou o método.

  **Sintaxe:** <nome_conjunto1>.containsAll(<nome_conjunto2>)

# HashSet e TreeSet

- Operação de União – Exemplo

```
public static void main(String[] args) {
    Set<String> clientes = new TreeSet<>(Arrays.asList("João", "Pedro",
                                                       "Paulo", "Maria", "Carlos"));

    Set<String> fornecedores = new TreeSet<>(Arrays.asList("Ana", "Paulo",
                                                           "Antonio", "Isabel", "Maria"));

    System.out.println("Clientes: " + clientes);
    System.out.println("Fornecedores: " + fornecedores);

    clientes.addAll(fornecedores);
    System.out.println("Clientes U Fornecedores: " + clientes);
}
```

```
Clientes: [Carlos, João, Maria, Paulo, Pedro]
Fornecedores: [Ana, Antonio, Isabel, Maria, Paulo]
Clientes U Fornecedores: [Ana, Antonio, Carlos, Isabel, João, Maria, Paulo, Pedro]
```

# HashSet e TreeSet

- Operação de Intersecção – Exemplo

```
public static void main(String[] args) {
    Set<String> clientes = new TreeSet<>(Arrays.asList("João", "Pedro",
                                        "Paulo", "Maria", "Carlos"));

    Set<String> fornecedores = new TreeSet<>(Arrays.asList("Ana", "Paulo",
                                        "Antonio", "Isabel", "Maria"));

    System.out.println("Clientes: " + clientes);
    System.out.println("Fornecedores: " + fornecedores);

    clientes.retainAll(fornecedores);
    System.out.println("Clientes n Fornecedores: " + clientes);
}
```

```
Clientes: [Carlos, João, Maria, Paulo, Pedro]
Fornecedores: [Ana, Antonio, Isabel, Maria, Paulo]
Clientes n Fornecedores: [Maria, Paulo]
```

# HashSet e TreeSet

- Operação de Diferença entre Conjuntos – Exemplo

```
public static void main(String[] args) {
    Set<String> clientes = new TreeSet<>(Arrays.asList("João", "Pedro",
                                                       "Paulo", "Maria", "Carlos"));

    Set<String> fornecedores = new TreeSet<>(Arrays.asList("Ana", "Paulo",
                                                           "Antonio", "Isabel", "Maria"));

    System.out.println("Clientes: " + clientes);
    System.out.println("Fornecedores: " + fornecedores);

    clientes.removeAll(fornecedores);
    System.out.println("Clientes - Fornecedores: " + clientes);
}
```

```
Clientes: [Carlos, João, Maria, Paulo, Pedro]
Fornecedores: [Ana, Antonio, Isabel, Maria, Paulo]
Clientes - Fornecedores: [Carlos, João, Pedro]
```

# HashSet e TreeSet

- Verificação de Subconjuntos – Exemplo

```
public static void main(String[] args) {
    Set<String> conjunto1 = new TreeSet<>(Arrays.asList("João", "Paulo",
                                          "Antonio", "Maria"));

    Set<String> conjunto2 = new TreeSet<>(Arrays.asList("Paulo", "Maria"));

    System.out.println("Conjunto 1: " + conjunto1);
    System.out.println("Conjunto 2: " + conjunto2);

    if (conjunto1.containsAll(conjunto2))
        System.out.println("O conjunto 1 contém o conjunto 2.");
}
```

```
Conjunto 1: [Antonio, João, Maria, Paulo]
Conjunto 2: [Maria, Paulo]
O conjunto 1 contém o conjunto 2.
```

# Mapas de Objetos

- Mapas de objetos são similiares às listas de objetos, com a diferença de que suas posições não precisam ser representadas por números inteiros.
- Cada elemento de um mapa é representado por um par de objetos denominados **chave** e **valor**.
- Da mesma forma que nas listas, nos mapas de objetos os elementos podem se repetir. Porém, não são permitidas chaves repetidas.

*l.bertholdo@ifsp.edu.br*

# Mapas de Objetos

- Os mapas de objetos são definidos por classes que implementam a interface **Map**, a qual declara métodos específicos para manipulação de mapas.
- Algumas das classes que implementam a interface **Map** visando a representação de mapas:
  - **HashMap** – Possui um mecanismo mais rápido nas operações de inserção e recuperação de objetos.
  - **TreeMap** – Possui um mecanismo mais lento, porém mantém a ordem das chaves de seus elementos.
- Ao criar uma instância destas classes, é preciso definir os tipos de objetos da chave e do valor a serem armazenados. Por exemplo:
  HashMap<**String**, **Integer**> mapa = new HashMap<>();

*l.bertholdo@ifsp.edu.br*

# HashMap e TreeMap

- Principais métodos:
  - **put:** Adiciona ou substitui um elemento no mapa.
    **Sintaxe:** <nome_mapa>.put(<chave>, <valor>)
  - **containsKey:** Retorna se um mapa contém uma determinada chave.
    **Sintaxe:** <nome_mapa>.containsKey(<chave>)
  - **containsValue:** Retorna se um mapa contém um determinado valor.
    **Sintaxe:** <nome_mapa>.containsValue(<valor>)
  - **get:** Retorna um elemento por meio de sua chave.
    **Sintaxe:** <nome_mapa>.get(<chave>)
  - **remove:** Remove um elemento do mapa por meio de sua chave.
    **Sintaxe:** <nome_mapa>.remove(<chave>)
  - **clear:** Remove todos os elementos do mapa.
    **Sintaxe:** <nome_mapa>.clear()

*l.bertholdo@ifsp.edu.br*

# HashMap e TreeMap

```java
public static void main(String[] args) {
    // Cria um mapa de objetos HashMap vazio.
    Map<String, String> proprietarios = new HashMap<>();

    // Adiciona os elementos ao mapa (proprietários de veículos).
    // A chave é a placa do veículo e o valor é o nome do proprietário.
    proprietarios.put("ABC1111", "João");
    proprietarios.put("ABC4444", "Maria");
    proprietarios.put("ABC3333", "Pedro");
    proprietarios.put("ABC2222", "Carlos");
    System.out.println(proprietarios); // Imprime os elementos do mapa.

    // Adiciona o elemento "Maria" novamente, porém em outra chave.
    proprietarios.put("ABC5555", "Maria");
    System.out.println(proprietarios); // Imprime os elementos do mapa.

    // Substitui o elemento "Pedro" por "Julia" na chave "ABC3333".
    proprietarios.put("ABC3333", "Julia");
    System.out.println(proprietarios); // Imprime os elementos do mapa.
}
```

```
{ABC1111=João, ABC2222=Carlos, ABC4444=Maria, ABC3333=Pedro}
{ABC1111=João, ABC2222=Carlos, ABC4444=Maria, ABC3333=Pedro, ABC5555=Maria}
{ABC1111=João, ABC2222=Carlos, ABC4444=Maria, ABC3333=Julia, ABC5555=Maria}
```

As chaves não são armazenadas em ordem crescente.

l.bertholdo@ifsp.edu.br

# HashMap e TreeMap

```java
public static void main(String[] args) {
    // Cria um mapa de objetos HashMap vazio.
    Map<String, String> proprietarios = new HashMap<>();

    // Adiciona os elementos ao mapa (proprietários de veículos).
    proprietarios.put("ABC1111", "João");
    proprietarios.put("ABC4444", "Maria");
    proprietarios.put("ABC3333", "Pedro");
    proprietarios.put("ABC2222", "Carlos");
    System.out.println(proprietarios); // Imprime os elementos do mapa.

    if (proprietarios.containsKey("ABC3333")) // Verifica se o mapa contém a chave "ABC3333".
        System.out.println("O mapa contém a placa 'ABC3333'.");

    if (proprietarios.containsValue("Maria")) // Verifica se o mapa contém o valor "Maria".
        System.out.println("O mapa contém o proprietário 'Maria'.");

    // Obtém o elemento que está na chave "ABC3333" no mapa.
    System.out.println("Proprietário do veículo placa ABC3333: " + proprietarios.get("ABC3333"));

    proprietarios.remove("ABC2222"); // Remove o elemento cuja chave é "ABC2222".
    System.out.println(proprietarios); // Imprime os elementos do mapa.
}
```

```
{ABC1111=João, ABC2222=Carlos, ABC4444=Maria, ABC3333=Pedro}
O mapa contém a placa 'ABC3333'.
O mapa contém o proprietário 'Maria'.
Proprietário do veículo placa ABC3333: Pedro
{ABC1111=João, ABC4444=Maria, ABC3333=Pedro}
```

l.bertholdo@ifsp.edu.br

# HashMap e TreeMap

- Principais métodos:
  - **size:** Retorna quantos elementos o mapa possui.
    **Sintaxe:** <nome_mapa>.size()
  - **isEmpty:** Retorna se um mapa está vazio.
    **Sintaxe:** <nome_mapa>.isEmpty()
  - **keyset:** Retorna as chaves dos elementos do mapa.
    **Sintaxe:** <nome_mapa>.keyset()
  - **values:** Retorna os valores dos elementos do mapa.
    **Sintaxe:** <nome_mapa>.values()

# HashMap e TreeMap

```java
public static void main(String[] args) {
    // Cria um mapa de objetos HashMap vazio.
    Map<String, String> proprietarios = new HashMap<>();

    // Adiciona os elementos ao mapa (proprietários de veículos).
    proprietarios.put("ABC1111", "João");
    proprietarios.put("ABC4444", "Maria");
    proprietarios.put("ABC3333", "Pedro");
    proprietarios.put("ABC2222", "Carlos");
    System.out.println(proprietarios); // Imprime os elementos do mapa.

    // Imprime o número de elementos contidos no mapa.
    System.out.println("Número de proprietários: " + proprietarios.size());

    if (!proprietarios.isEmpty()) // Verifica se o mapa está vazio.
        System.out.println("O mapa não está vazio.");

    // Imprime as chaves dos elementos do mapa.
    System.out.println("Placas: " + proprietarios.keySet());

    // Imprime os valores dos elementos do mapa.
    System.out.println("Proprietários: " + proprietarios.values());
}
```

```
{ABC1111=João, ABC2222=Carlos, ABC4444=Maria, ABC3333=Pedro}
Número de proprietários: 4
O mapa não está vazio.
Placas: [ABC1111, ABC2222, ABC4444, ABC3333]
Proprietários: [João, Carlos, Maria, Pedro]
```

l.bertholdo@ifsp.edu.br

# Referências


- H. M. Deitel; P. J. Deitel; Java Como Programar – 4ª Edição. Bookman, 2003.

- Rafael Santos; Introdução à Programação Orientada a Objetos usando Java – 2ª edição. Rio de Janeiro: Elsevier, 2013.