



C125/C206 – Programação Orientada a Objetos  
com Java

# Coleções no Java

Prof. Phyllipe Lima  
phyllipe@inatel.br





# Arrays, o Resgate

- ☕ Já sabemos criar **arrays** em Java. E também sabemos algumas desvantagens envolvidas na manipulação de **arrays**
  - ☕ Tamanho fixo. Não conseguimos mudar o seu tamanho em tempo de execução.
  - ☕ Precisamos varrer o **array** todo para procurar alguma posição vazia
    - ☕ Lembre-se dos métodos que criamos para inserir dados no **array**
  - ☕ Resumindo, precisamos criar métodos auxiliares para manipular o **array**





# A API de Coleções

- ☕ Para resolver os problemas relacionados a **arrays**, o Java possui a API de Coleções, ou como é mais conhecida, ***Collections API***.
- ☕ Já sabemos que API é basicamente, ***uma biblioteca de códigos disponíveis para utilizarmos*** em nossas próprias soluções. E não estamos preocupados em como essa funcionalidade ***está implementada*** queremos apenas utilizá-la!
- ☕ A ***Collection API*** nos fornece Classes e Interfaces para manipularmos coleções dos mais variados tipos de dados

# A API de Coleções



☕ Por exemplo, podemos manipular uma **coleção** de jogos de tabuleiro 😊





# A Interface *List*

- ☕ Relembrando os velhos tempos de Algoritmos II, aprendemos que uma **lista** é uma estrutura de dados onde podemos armazenar informações, e ela **cresce** conforme precisamos **inserir** mais elementos.
- ☕ Diferentemente dos arrays, seu tamanho não é fixo!
- ☕ No Java, utilizamos a interface **List** para criamos as nossas listas.
- ☕ Como **List** é uma **interface**, existem diversas classes que implementam essa **interface** como por exemplo a classe **ArrayList**.



# *ArrayList*

- ☕ A classe ***ArrayList*** implementa a interface ***List***.
- ☕ Existem outras classes, como a ***LinkedList***, que também implementa a interface ***List***.
- ☕ A implementação mais utilizada da interface ***List*** é a ***ArrayList***, que trabalha com um ***array*** interno, gerado dinamicamente. Portanto, ela é mais rápida na pesquisa do que sua concorrente, a ***LinkedList***, que é mais rápida na inserção e remoção de itens nas pontas.
- ☕ Não confundam a classe ***ArrayList*** com o que conhecemos de ***arrays***. Apesar de internamente ela usar esse conceito, ele tem seu tamanho definido dinamicamente.



- ☕ Desejamos utilizar a **interface List** do pacote **java.util.List**.
- ☕ Utilizaremos a implementação concreta **ArrayList**.
- ☕ Começaremos criando uma lista genérica, que aceita **todo** tipo de dado possível que o Java trabalha. Isto é, qualquer **Object**. A classe mãe de todas as classes do Java. Observe também que não definimos o tamanho dessa lista em nenhum momento.

```
List listaGenerica = new ArrayList();  
  
listaGenerica.add(1);  
listaGenerica.add("Capiroto");  
listaGenerica.add("String");  
listaGenerica.add(45);
```

- ☕ Para adicionarmos elementos nessa lista, utilizamos o método **add(elemento)**





- ☕ O método ***add()***, possui uma sobrecarga onde podemos passar também a posição onde desejamos adicionar o elemento.
- ☕ Mas tome cuidado, pois precisamos garantir que ***de fato*** existe essa posição, ou teremos uma ***IndexOutOfBoundsException***
- ☕ A primeira posição possui índice ***zero (0)***.

```
List listaGenerica = new ArrayList();
```

```
listaGenerica.add(1);  
listaGenerica.add("Capiroto");  
listaGenerica.add("String");
```

```
//Não estamos apagando o que se encontra no índice 1.  
//Estamos inserindo a String Capiroto2 no índice 1.  
//E deslocando a lista para frente.  
//Ou seja, adicionando um novo elemento  
listaGenerica.add(1, "Capiroto2");
```





☕ Existe uma desvantagem em criarmos listas que aceitam todo o tipo de dado. Para fazer a recuperação precisamos fazer um **cast** explícito para o tipo de dado que desejamos.

☕ Para buscar dados na lista usamos o método **get()** passando o índice da posição.

```
listaGenerica.add(1);  
listaGenerica.add("Capiroto");  
listaGenerica.add("String");
```

```
int numero = (int) listaGenerica.get(0);|
```

☕ No exemplo buscamos o elemento na posição 0, isto é, o primeiro elemento.

☕ Observe que também foi necessário fazer um **cast** para **int**.



- ☕ Utilizar listas genéricas podem ser problemáticas, e levar **exceções** relacionadas a **cast**.
- ☕ É recomendado que utilizemos listas para tipos específicos, e o Java oferece recursos para isso.
- ☕ Dessa forma o próprio compilador irá nos proteger de adicionarmos elementos inadequados.

```
List<String> listaDeString = new ArrayList<String>();
```

```
listaDeString.add(1); // Nao compila, pois a lista é para String  
listaDeString.add("Capioto");  
listaDeString.add("String");
```



☕ Para buscarmos um elemento específico, não precisamos mais fazer o ***cast***.

☕ O compilador irá nos avisar se tentarmos atribuir um elemento da lista para um tipo inadequado.

```
List<String> listaDeString = new ArrayList<String>();
```

```
listaDeString.add("Capiroto");
```

```
listaDeString.add("String");
```

```
String elemento = listaDeString.get(0); // Compila :)|
```

```
int elemento2 = listaDeString.get(1); // Não compila
```

☕ Podemos iterar na lista utilizando *for each* ou o *for* tradicional



```
List<String> listaDeString = new ArrayList<String>();|
```

```
listaDeString.add("Capiroto");  
listaDeString.add("String");  
listaDeString.add("Black Knight");
```

```
//for each
```

```
for (String varTemporaria : listaDeString) {  
    System.out.println(varTemporaria);  
}
```

```
//for tradicional
```

```
for (int i = 0; i < listaDeString.size(); i++) {  
    System.out.println(listaDeString.get(i));  
}
```



☕ Podemos também criar uma lista de elementos ***de um tipo criado pelo programador.***

☕ Isto é, nossas próprias Classes e Interfaces.

☕ Considere uma lista para elementos ***do tipo*** Inimigo

```
List<Inimigo> listaInimigos = new ArrayList<>();|
```

```
Inimigo inimigo1 = new Inimigo("Black Knight", 150);  
Inimigo inimigo2 = new Inimigo("Silver Knight", 200);
```

```
listaInimigos.add(inimigo1);  
listaInimigos.add(inimigo2);
```



# Ordenando Elementos

- ☕ Novamente, resgatando os velhos tempos de Algoritmos II, vimos diversos algoritmos de ordenação. E fizemos algumas implementações
- ☕ O Java possui a classe ***Collections*** que tem um método estático ***sort(List<T>)***, capaz de ordenar uma lista de elementos armazenadas em uma ***List***.
- ☕ Para serem ordenados, os dados precisam ser do tipo de uma classe que ***implementa*** a interface ***Comparable***.
- ☕ Isto é, precisam ser dados comparáveis.
- ☕ Algumas classes do Java já são comparáveis, isto é, em sua definição elas implementam ***a interface Comparable*** e o método ***compareTo()***.
- ☕ Veremos alguns exemplos



```
List<String> listaNomes = new ArrayList<String>();
```

```
listaNomes.add("Capiroto");  
listaNomes.add("Apolonio");  
listaNomes.add("Badumtin");
```

```
Collections.sort(listaNomes);
```

```
for (String nome : listaNomes) {  
    System.out.println(nome);  
}
```

☕ Ao passarmos a lista ***listaNomes*** para o método ***sort()***, a lista ficará ordenada de forma lexicográfica

```
<terminated> Main (7) [Java Application]
```

```
Apolonio  
Badumtin  
Capiroto
```





- ☕ E uma lista de inteiros?
- ☕ A **interface List** não aceita tipos primitivos, apenas classes.
- ☕ Para resolver isso o Java possui uma categoria de classes chamadas **Classes Wrappers**
- ☕ O objetivo dessas classes é envolver (**wrap**) os tipos primitivos.
- ☕ Para o tipo **int** temos a classe **Integer**.
- ☕ É muito simples utilizar as classes **wrappers**, uma vez que o Java faz tudo praticamente sozinho nos bastidores
- ☕ Temos também a classe **Double, Float e Char**.



```
List<Integer> listaInteiros = new ArrayList<>();  
listaInteiros.add(34);  
listaInteiros.add(5);  
listaInteiros.add(56);  
listaInteiros.add(23);
```

```
Collections.sort(listaInteiros);
```

```
//Aqui usamos int
```

```
for (int numero : listaInteiros) {  
    System.out.println(numero);  
}
```

```
//Mesmo funcionamento na leitura dos dados
```

```
//int ou Integer
```

```
for (Integer numero : listaInteiros) {  
    System.out.println(numero);  
}
```

5
23
34
56

5
23
34
56

☕ Podemos salvar dados do tipo **Integer** em variáveis do tipo **int** sem a necessidade de **casting**. O Java faz isso automaticamente

# E Para Ordenar Nossas Próprias Classes?



- ☕ Vamos ordenar uma lista de dados **do tipo** Inimigo, usando como chave de comparação a vida. Podemos utilizar qualquer campo de comparação.
- ☕ Precisamos fazer a nossa classe Inimigo **implementar** a interface **Comparable** e sobrescrever o método **compareTo**.
- ☕ A interface **Comparable** é parametrizada. Então devemos passar, entre **< >**, qual classe podemos ser comparados.
- ☕ No nosso exemplo queremos comparar Inimigo com outro Inimigo



```
public class Inimigo implements Comparable<Inimigo> {  
  
    private String nome;  
  
    private int vida;  
  
    public Inimigo(String nome, int vida) {  
        this.nome = nome;  
        this.vida = vida;  
    }  
  
    @Override  
    public int compareTo(Inimigo o) {  
        return 0;  
    }  
}
```



- ☕ Vamos entender o método ***compareTo()***
- ☕ No nosso exemplo ela recebe um parâmetro ***do tipo*** Inimigo, que iremos utilizar para comparar com a nossa instância (***this***).
- ☕ Se desejamos fazer ordenação crescente, basta devolver um valor ***negativo*** caso a nossa instância (***this***) seja menor que o Inimigo recebido por parâmetro. E passamos um valor positivo caso nossa instância (***this***) seja maior.
- ☕ Se forem iguais devolvemos zero (0)
- ☕ Para fazer ordenação decrescente basta fazer a lógica acima ao contrário.



```
public class Inimigo implements Comparable<Inimigo> {  
  
    private String nome;  
  
    private int vida;  
  
    public Inimigo(String nome, int vida) {  
        this.nome = nome;  
        this.vida = vida;  
    }  
  
    @Override  
    public int compareTo(Inimigo o) {  
        if(this.vida < o.getVida())  
            return -1;  
        if(this.vida > o.getVida())  
            return 1;  
        return 0;  
    }  
}
```



```
Inimigo inimigo1 = new Inimigo("Black Knight", 150);  
Inimigo inimigo2 = new Inimigo("Monstro", 50);  
Inimigo inimigo3 = new Inimigo("Silver Knight", 200);
```

```
listaInimigos.add(inimigo1);  
listaInimigos.add(inimigo2);  
listaInimigos.add(inimigo3);
```

```
for (Inimigo inimigo : listaInimigos) {  
    System.out.println(inimigo.getNome() + " : "+  
        inimigo.getVida() );  
}
```

```
Collections.sort(listaInimigos);
```

//Depois de ordenar

```
for (Inimigo inimigo : listaInimigos) {  
    System.out.println(inimigo.getNome() + " : "+  
        inimigo.getVida() );  
}
```

Black Knight : 150
Monstro : 50
Silver Knight : 200

Monstro : 50
Black Knight : 150
Silver Knight : 200





# Exercício 1

- ☕ Crie uma lista de números do tipo ***double***, e preencha ao menos cinco (5) valores.
- ☕ Faça a ordenação decrescente e imprima esses valores utilizando um ***foreach (use os três for)***



## Exercício 2 - Desafio

- ☕ Crie uma classe **abstrata** `Inimigo` com os parâmetros `nome(String)` e `vida (int)`. Essa classe deve implementar a interface **`Comparable<Inimigo>`** e ter um construtor com os dois parâmetros
- ☕ Como a classe é abstrata você não precisa implementar o método **`compareTo`**. Mas mesmo assim o faça.
- ☕ Crie três classes filhas de `Inimigo`: `BlackKnight`, `SilverKnight` e `FireKnight`.
- ☕ Na ***main*** crie uma lista de `Inimigo`, com uma instância de cada classe filha.
- ☕ Ordene pela vida de forma decrescente!



# Mais da Classe Collections

☕ Através da classe ***Collections*** podemos, além de ordenar, realizar outras funções interessantes com List

☕ ***max(List)*** retorna o maior elemento da lista

☕ ***min(List)*** retorna o menor elemento da lista

☕ ***binarySearch(List,elemento)*** faz uma busca binária e retorna a posição do elemento ou um número negativo caso não o encontre. ***Importante: A lista precisa estar ordenada para usar a busca binária.*** Caso contrário os resultados são imprevisíveis.

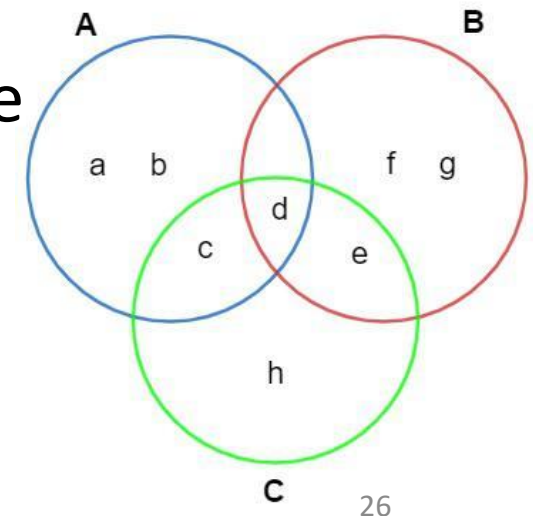
☕ ***reverse(List)***. Cria a lista de forma reversa.

☕ A lista precisa ser de elementos que implementam a interface ***Comparable***

# Set



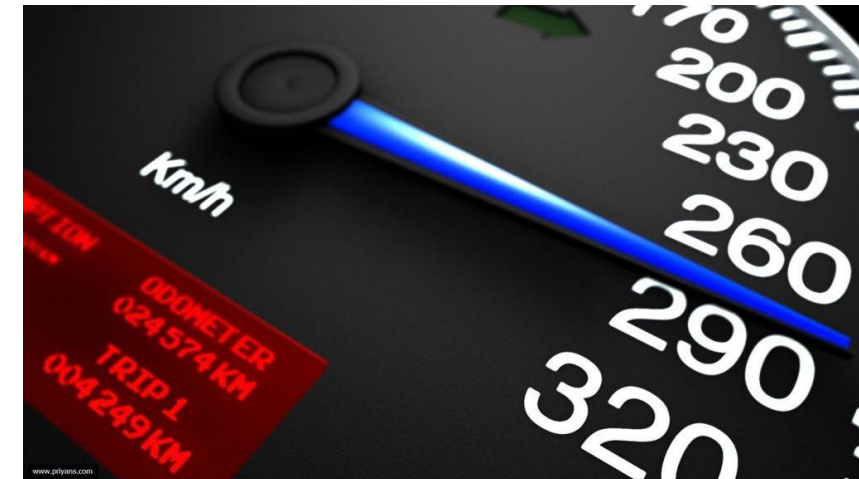
- ☕ O Java possui uma interface ***java.util.Set*** para a criação de uma lista em forma de Conjunto.
- ☕ Assim como na matemática, em coleções do tipo ***Set***, não é permitida a presença de elementos duplicados.
- ☕ ***Sets*** também não apresentam o conceito de “posição”, assim não é possível buscar um elemento pelo seu índice.
- ☕ A interface não garante a ordem aos elementos. Variando de implementação para implementação.



# Set



- ☕ Os **Sets** podem não parecer interessantes. Porém, existem implementações, como o **HashSet**, que possui alto desempenho para buscar elementos.
- ☕ Também é útil quando se deseja guardar uma lista de elementos e precisamos garantir que não haverá duplicidade.





# HashSet - Criação

- ☕ Vamos criar um conjunto com a **HashSet**.
- ☕ Essa implementação utiliza um **tabela hash** para guardar os elementos e não garante a ordem.

```
//Criando nosso conjunto de Strings  
Set<String> conjunto = new HashSet<String>();
```

```
conjunto.add("Malfurion_1");//Adicionando elementos  
conjunto.add("Malfurion_2");  
conjunto.add("Malfurion_3");  
conjunto.add("Malfurion_4");  
conjunto.add("Malfurion_5");
```



# HashSet - Percorrer

☕ Como os elementos dos **Sets** não possuem índices, podemos utilizar o **foreach** para percorrer os elementos.

```
//Criando nosso conjunto de Strings
Set<String> conjunto = new HashSet<String>();

conjunto.add("Malfurion_1");//Adicionando elementos
conjunto.add("Malfurion_2");
conjunto.add("Malfurion_3");
conjunto.add("Malfurion_4");
conjunto.add("Malfurion_5");
```

```
for (String elemento : conjunto) {
    System.out.println(elemento);
}
```

Malfurion\_3  
Malfurion\_2  
Malfurion\_5  
Malfurion\_4  
Malfurion\_1





# HashSet - Remoção

☕ Para removermos um elemento usamos o método ***remove(elemento)*** e passamos o elemento que desejamos remover. Caso não exista, a função retorna ***false***. Se a remoção ocorrer como sucesso, a função retorna ***true***.

```
System.out.println(conjunto.remove("Malfurion_5"));
```

```
for (String elemento : conjunto) {  
    System.out.println(elemento);  
}
```

true

Malfurion\_3  
Malfurion\_2  
Malfurion\_4  
Malfurion\_1



# HashSet – Usando Iterator

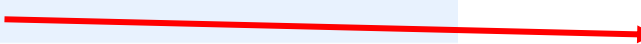
- ☕ Antes do **foreach** era necessário utilizar a interface **Iterator** para percorrer um **Set**. Nos bastidores é isso que o **foreach** faz.
- ☕ Mas o **Iterator** pode ser útil se desejarmos percorrer um **Set** e, ao identificar algum elemento, fazer a remoção deste de forma segura.
- ☕ Vamos remover, por exemplo, “Malfurion\_5”
- ☕ Perceba que, nesse caso, faremos a remoção pelo objeto **Iterator** e não pelo “conjunto” (igual no exemplo anterior)



```
Iterator<String> it = conjunto.iterator();

//O metodo hasNext() devolve se existe elemento.
//Enquanto existir vamos percorrer o "conjunto"
while(it.hasNext()) {
    if(it.next() == "Malfurion_5") {
        //Irá remover o último elemento
        //retornado pelo Iterator
        //Repare que fazemos a remoção pelo Iterator
        //E não diretamente no conjunto
        it.remove();
    }
}

for (String elemento : conjunto) {
    System.out.println(elemento);
}
```

A red arrow originates from the line `System.out.println(elemento);` in the code block and points towards the list of elements.

Malfurion_3
Malfurion_2
Malfurion_4
Malfurion_1




# LinkedHashSet

- ☕ O **LinkedHashSet** mantém a ordem de inserção, mas perde um pouco em desempenho se comparado ao **HashSet**.
- ☕ Se não é necessário manter a ordem de inserção dê preferência ao **HashSet**.

```
//Criando LinkedHashSet|
Set<String> conjuntoOrdem = new LinkedHashSet<String>();
conjuntoOrdem.add("Malfurion_1");
conjuntoOrdem.add("Malfurion_2");
conjuntoOrdem.add("Malfurion_3");
conjuntoOrdem.add("Malfurion_4");

for (String elemento : conjuntoOrdem) {
    System.out.println(elemento);
}
```



```
Malfurion_1
Malfurion_2
Malfurion_3
Malfurion_4
```



# Mapas – Chave->Valor

- ☕ Imagine que você deseja salvar elementos no formato de um dicionário, onde dada uma palavra (chave) você pegue o seu significado (valor).
- ☕ O Java oferece uma interface ***java.util.Map*** onde é possível salvar elementos nesse formato <chave,valor>.
- ☕ Para salvar dados no Mapa usamos o método ***put (chave, valor)***.
- ☕ A busca nessa estrutura é bastante rápida
- ☕ No Java, o nome ***Map*** leva a ideia de ***mapear um valor***
- ☕ Depois de inserido no ***Map***, podemos buscar um elemento através do método ***get(Chave)***.
- ☕ Como curiosidade, no C# a interface se chama ***Dictionary***.



# HashMap

- ☕ Utilizaremos a implementação concreta **HashMap**. Vamos criar um mapa de quebra-cabeça, para armazenar o nome e a quantidade de peças
- ☕ A **chave** será o nome do quebra-cabeças e o **valor** será o número de peças. Assim, nossa chave será do tipo **String** e nosso valor do tipo **Integer** (lembre-se das classes **wrappers**).

```
//Criando o mapa
Map<String, Integer> mapaPuzzle = new HashMap<String, Integer>();

//Adicionando valores
mapaPuzzle.put("Jardim Vitoriano", 5000);
mapaPuzzle.put("Expresso Noturno", 5000);
mapaPuzzle.put("As Quatro Estações", 4000);
mapaPuzzle.put("Le Petit Café", 6000);
```



# HashMap – Buscando Valores

- ☕ Com nosso mapa criado podemos buscar valores. Imagine que queremos saber quantas peças o puzzle Jardim Vitoriano possui.
- ☕ Basta buscarmos pela chave “Jardim Vitoriano”

```
//Salvamos em uma variável do tipo Integer  
//Pois se o mapa não encontrar a chave, será devolvido null  
Integer numPecas = mapaPuzzle.get("Jardim Vitoriano");  
if(numPecas != null)  
    System.out.println(numPecas);
```





# HashMap – Sobrescrevendo

☕ Só pode existir um único valor para cada chave. Portanto se fizermos uma inserção com uma chave já existe, o valor será sobrescrito.

```
Integer numPecas = mapaPuzzle.get("Jardim Vitoriano");  
if(numPecas != null)  
    System.out.println(numPecas);
```

```
mapaPuzzle.put("Jardim Vitoriano", 1000);  
//A chave Jardim Vitoriano já existe. Então o valor será  
//sobrescrito
```

```
numPecas = mapaPuzzle.get("Jardim Vitoriano");  
if(numPecas != null)  
    System.out.println(numPecas);
```


5000
1000



# HashMap – Iterando com Lambda

- ☕ Iterar sobre um Mapa não é tão simples quanto em listas e conjuntos, mas também não é complicado.
- ☕ Com o Java 8 podemos utilizar um método chamado ***forEach()*** presente na interface ***Collection***. Combinado com as ***expressões lambda*** (discutiremos em outro momento) fica bem simples iterar sobre um mapa.

```
mapaPuzzle.forEach((chave, valor) -> {  
    System.out.println("Chave: " + chave);  
    System.out.println("Valor: " + valor);  
});
```



```
Chave: As Quatro Estações  
Valor: 4000  
Chave: Espresso Noturno  
Valor: 5000  
Chave: Jardim Vitoriano  
Valor: 5000  
Chave: Le Petit Café  
Valor: 6000
```

# Exercício 3 – Medindo Desempenho




- ☕ Nesse exercício iremos medir o desempenho de um ***ArrayList***, ***HashSet*** e ***HashMap***.
- ☕ Crie essas três estrutura e preencha com 100000 (Cem mil) valores inteiros. A chave do Mapa também deverá ser inteira (Integer)
- ☕ Meça o tempo gasto para inserir cada elemento.
- ☕ Após as coleções preenchidas, busque todos os elementos e meça o tempo gasto em cada busca.
- ☕ Crie variáveis auxiliares para realizar tarefa.
- ☕ Utilize o método `System.currentTimeMillis()` para buscar a hora do sistema no instante.



# Vídeo Aula

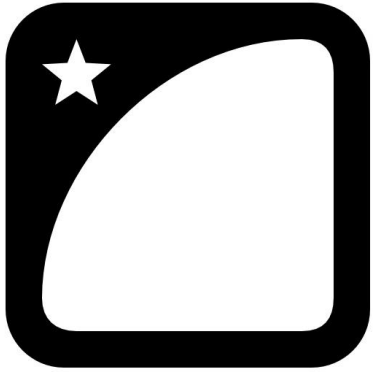


 <https://youtu.be/8aQusNoOzB4>

 OBS (2020/1)



# Material Complementar

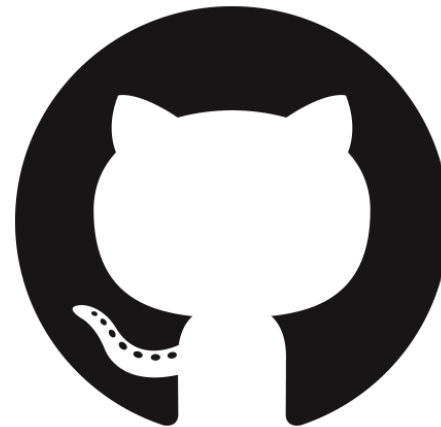


☕ Capítulo 15 da apostila FJ-11  
☕ Até o item 15.5



# Resolução dos Exercícios

<https://github.com/phillima-inatel/C125>





C125/C206 – Programação Orientada a Objetos  
com Java

# Coleções no Java

Prof. Phyllipe Lima  
phyllipe@inatel.br

