# Visão Geral de *Generics* e *Collections*

Prof. Jeferson Souza, MSc. (thejefecomp)

jeferson.souza@udesc.br

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

JOINVILLE
CENTRO DE CIÊNCIAS
TECNOLÓGICAS

# Introdução aos Generics e as Collections

## Olha lá professor, o que são os famosos Generics e Collections?

Ao que tudo indica, até este exato momento, o conhecimento a respeito dos *Generics* e das *Collections* é muito incipiente, a considerar a probabilidade de não se ter tido consciência de suas utilizações. Os famosos *Generics* e *Collections* são fundamentais na Linguagem Java, a constituirem formas de representação de tipos de maneira generalizada (i.e. *Generics*), e fornecimento de diferentes estruturas de dados e algoritmos associados às referidas estruturas (i.e. *Collections*).

# Introdução aos Generics e às Collections

## Prazer, Generics, somos os shapeshifters da Linguagem Java

Sua definição é generalizada, a compor uma característica importante da Linguagem Java na especificação de modelos e escrita de código-fonte independente do tipo de dado utilizado. Os *Generics* foram introduzidos na linguagem Java a partir da versão 1.5 (Java 5 para os mais íntimos), e o seu principal objetivo é permitir a referida generalização de uma forma distinta da alcançada por meio de uma hierarquia de Classes ou *Interfaces* tradicional, a ser o compilador responsável pela automatização.

# Introdução aos Generics e às Collections

## Prazer, Collections, as “caixinhas” (i.e. tupperwares) da Linguagem Java

As coleções especificam formas distintas de armazenar e manipular dados, a representar implementações de algoritmos e estruturas de dados disponibilizadas pela Linguagem Java para o desenvolvimento de programas.

# Generics

## Como funcionam os Generics?

Ao entrar-se nos detalhes que compõe o domínio dos tipos genéricos da Linguagem Java, fica-se frente a frente com um conceito aplicado no domínio das linguagens de programação chamado de **Type Erasure** [GoslingEtAl, 2021]. De forma simplificada, **Type Erasure** pode ser definido pela operação de substituição de tipos necessária para o bom andamento do programa, a representar a troca de um tipo por outro de acordo com o contexto e domínio associado à referida troca. No caso dos *Generics*, o uso de **Type Erasure** permite a troca de um tipo de dados genérico por um outro tipo de dados concreto (que pode ser inclusive o *Object*) no momento da compilação do programa.

# Generics

**Mas se os tipos são substituídos, não é mais fácil usar diretamente o tipo concreto?**

No momento que tem-se a necessidade de especificar um dado comportamento que pode ser aplicado a um conjunto inimaginável de elementos, o uso dos *Generics*, alcançado pela automatização fornecida pelo compilador da Linguagem Java, torna-se obrigatório. As *Collections* são o principal exemplo de uso de *Generics* na Linguagem Java. Todas as estruturas de dados disponíveis podem ser parametrizadas com qualquer tipo definido na Linguagem, ou diretamente pelo programa desenvolvido.

# Generics - Convenção de Nomes

## Como definir o nome de um tipo de dado genérico?

Teoricamente, um tipo de dado genérico da Linguagem Java pode receber qualquer nome que seja do agrado do projetista do modelo e/ou do desenvolvedor do código-fonte, desde que o nome não viole nenhuma restrição imposta pela linguagem. Entretanto, existem algumas convenções de nomes seguidas pela Linguagem Java que tornam mais simples a identificação e utilização dos *Generics*. Vamos a elas!

# Generics - Convenção de Nomes

## Convenção de nomes para tipos genéricos [Boyarsky&Selikoff, 2016]

- **E** para elemento;
- **K** para uma chave em um mapa;
- **V** para um valor em um mapa;
- **N** para um número;
- **T** para um tipo de dado genérico;
- **S,U,V**, e assim por diante, para múltiplos tipos de dados genéricos;

# Generics

**Para generalizar basta parametrizar...**

Para definir qualquer elemento que use um tipo de dado genérico, basta parametrizar o referido elemento. *Interfaces*, Classes, e Métodos podem ser parametrizados para que permitam o uso dos *Generics*. Essa parametrização é realizada diretamente no código-fonte, por meio do uso do operador *diamond* na definição da *Interface*, Classe, ou Método onde o tipo de dado genérico vai ser especificado pela primeira vez.

# Generics - Exemplo de Definição de Interface Genérica

## Definição de Interface Genérica

```
public interface Controle<T>{
    boolean ligar(T equipamento);
    boolean desligar(T equipamento);
    ...
}
```

## Tá, então o T é o tipo genérico?

Exatamente! O *T* é o tipo genérico utilizado na especificação da *Interface Controle*. Como o tipo *T* foi definido diretamente na especificação da *Interface* Controle, pode-se utilizá-lo livremente dentro do escopo da referida *Interface*.

# Generics - Exemplo de Concretização de Interface Genérica

## Concretizar Interface Genérica com tipo Concreto

```
public class ControleImpl implements Controle<ArCondicionado>{
  public boolean ligar(ArCondicionado equipamento){
    ...
  }
  public boolean desligar(ArCondicionado equipamento){
  ...
  }
}
```

A especificação do tipo concreto só precisa ser fornecida na implementação da *Interface*

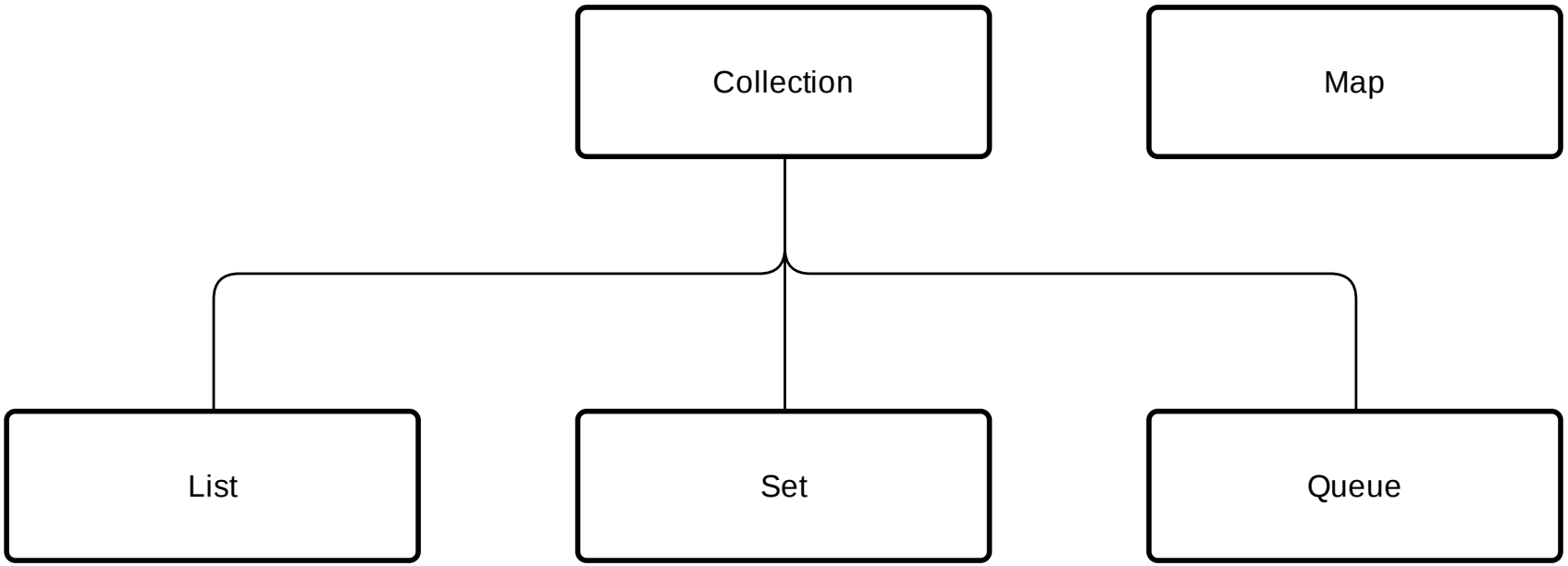
# Generics - Exemplo de Concretização de Interface Genérica

## Concretizar Interface Genérica com tipo Genérico

```
public class ControleImpl<T> implements Controle<T>{
  public boolean ligar(T equipamento){
    ...
  }
  public boolean desligar(T equipamento){
  ...
  }
}
```

A definição do tipo genérico precisa ser estendida para a especificação da classe *ControleImpl*.

# Generics - As Interfaces Genéricas das Collections




As Interfaces Genéricas da Linguagem Java presentes nas *Collections* [Boyarsky&Selikoff, 2016]

# Generics - Exemplo de Extensão de Interface Genérica

## Olha a List ai gente!

```
public interface List<E> extends Collection<E>{
   ...
}
```

## Eeeee lá! Então o E é o tipo genérico de List?

Exatamente! O *E* é o tipo genérico utilizado na especificação da *Interface List*. Percebe-se que o tipo *E* é oriundo da *Interface* Collection, e como não recebeu um tipo concreto em *List*, sua definição precisa ser estendida para *List*.

# Generics - Exemplo de Definição de Classe Genérica

## Definição de Classe Genérica

```
public class Aquecedor<E>{
  public boolean aquece(E entidade, Float
temperaturaAlvo){
    ...
  }
  public void arrefece(E entidade){
    ...
  }
}
```

A definição do tipo genérico está presente diretamente na especificação da classe *Aquecedor*.

# Generics - Exemplo de Definição de Método Genérico

## Definição de Método Genérica

**public** <T> **boolean realizarComputacao(T tipo){...}**

No caso da definição de métodos com *Generics*, o tipo genérico é definido juntamente com a especificação do método, a indicar um tipo que é independente da especificação da classe onde o método é declarado.

# Generics - Exemplo de Definição de Método Genérico

**Definição de Método Estático Genérica**

```
public static <T> boolean realizarComputacao(T tipo){...}
public static <T> T getEntidade(Long id){...}
```

A única forma de uso de *Generics* com métodos estáticos é definir o tipo genérico juntamente com a especificação do método, a indicar, portanto, um tipo que é independente da especificação da classe onde o método é declarado.

# Generics - Limites de tipos

## Limites de tipos

Até o momento foi visto que pode-se definir tipos genéricos, que após a realização de *Type erasure* podem resultar na substituição dos referidos tipos até mesmo pela classe Object, a não oferecer muitos limites naquilo que pode ser aceito por uma dada referência de Classe, *Interface*, ou ainda parâmetros de um dado método. Para tal, existem os limites de tipos, os quais podem ser especificados por meio do *wildcard* "**?**".

# Generics - Limites de tipos

## Antes de saber o que é o wildcard "?"...

Os tipos genéricos "sem limites" que não são especificados com o wildcard "?", tal como na definição:

**public class Aquecedor<E>{ ... }**

são verificados em tempo de compilação, a ter a clara garantia que serão substituidos por tipos concretos disponíveis no momento da execução do programa. Esses tipos concretos que o compilador tem a certeza que estarão disponíveis em tempo de execução são chamados de **Reifiable types** [GoslingEtAl, 2021], i.e., toda a classe, interface, e derivados presentes no *classpath* do programa em tempo de compilação.

# Generics - Limites de tipos

## O wildcard "?"...

O *wildcard* "**?**" representa um tipo genérico desconhecido, o qual não se consegue saber nada a respeito até o momento da execução do programa. É esse tipo genérico desconhecido que é utilizado como base para especificação dos limites de tipos genéricos.

# Generics - Limites de tipos

## Três formas de uso do wildcard "?"...

Existem três (3) formas de uso do *wildcard* "**?**" [Boyarsky&Selikoff, 2016]:

| **Tipo de limite** | **Sintaxe** | **Exemplo** |
|---|---|---|
| *wildcard* sem limite | **?** | List<?> lista = **new** ArrayList<String>(); |
| *wildcard* com limite superior | **? extends tipo** | List<? **extends** Exception> lista = **new** ArrayList<RuntimeException>(); |
| *wildcard* com limite inferior | **? super tipo** | List<? **super** Exception> lista = **new** ArrayList<Object>(); |

# Generics - Limites de tipos

## O wildcard "?" sem limite (Imutável)...

O *wildcard* "**?**" sozinho pode representar qualquer tipo de dado disponível no momento da execução do programa. Ao utilizar o wildcard "**?**" sem limites, o desenvolvedor informa ao compilador da linguagem Java que qualquer tipo de dado pode ser aceito. **Detalhe: o wildcard "?" sem limite é imutável**. Exemplo [Boyarsky&Selikoff, 2016]:

```
public static void imprimeLista(List<?> lista){
  for (Object x : lista) System.out.println(x);
}
```

O método *imprimeLista()* aceita imprimir qualquer tipo de lista: List<Integer>, List<String>, e até mesmo List<Object>. Percebam que List<Integer> é um tipo diferente de List<Object>, e é por essa razão que deve-se utilizar o *wildcard* "?" para poder aceitar qualquer tipo de lista.

# Generics - Limites de tipos

## O wildcard "?" com limite superior (Imutável)...

O *wildcard* "**?**" utilizado com limite superior usa o poder hierárquico fornecido pela linguagem Java para limitar a "classe pai" de um tipo de dado aceito pela definição de Classe, *Interface* ou método especificado. Para tal, a palavra reservada **extends** indica a Classe/*Interface* utilizada como limite superior. **Detalhe: o wildcard "?" com limite superior é imutável**. Exemplo [Boyarsky&Selikoff, 2016]:

```
public static long total(List<? extends Number> lista){
   long contador = 0;
   for (Number numero : lista){
     contador += numero.longValue();
     return contador;
}
```

O método *total()* aceita contar listas de números, i.e. List<Integer>, List<Float>, etc.

# Generics - Limites de tipos

## O wildcard "?" com limite inferior...

O *wildcard* "**?**" utilizado com limite inferior também utiliza o poder hierárquico fornecido pela linguagem Java. No caso dos limites inferiores têm-se um poder adicional: as definições de Classes, *Interfaces*, e métodos aceitam a especificação de **atributos/variáveis/parâmetros que podem ser modificados**. Para tal, a palavra reservada **super** indica a classe/interface utilizada como limite inferior. Exemplo [Boyarsky&Selikoff, 2016]:

```
public static long adicionaSom(List<? super String> lista){
  lista.add("quack");
}
```

O método *adicionaSom()* aceita adicionar qualquer objeto que possa ser utilizado como um objeto do tipo String. Isso implica que caso o limite inferior contenha classes herdeiras, qualquer classe herdeira poderá ser adicionada.

# Generics - Limites de tipos

## Detalhes importantes sobre o uso do wildcard "?"...

- O uso de *wildcard* **sem limite** ou **com limite superior** implica na **definição de atributos/variáveis/parâmetros imutáveis**;
- Não se pode substituir o *wildcard* "?" por qualquer outro tipo, a incluir um tipo genérico. Exemplo: <C **super String**> não pode substituir <? **super String**>;
- No caso dos limites inferiores, os quais não são imutáveis, só se pode realizar operações que modificam tipos de dados da Classe/*Interface* especificada pelo limite em questão, ou suas classes herdeiras.

# Generics - Limites de tipos (Continuação)

## Detalhes importantes sobre o uso do wildcard "?"...

- Ainda em relação aos limites inferiores, aceita-se a atribuição de, por exemplo, qualquer estrutura de dado especificada com base em Classes/*Interfaces* que estejam em posições hierárquicas superiores à da Classe/Interface que representa o limite inferior. Entretanto, somente Classes/*Interfaces* que fizerem parte da hierarquia onde o limite inferior for a Classe/*Interface* pai poderão ser adicionadas à referida estrutura.

## Detalhe importante dos limites inferiores...

```
List<? super String> lista = new ArrayList<String>();
List<? super String> listaStr = new ArrayList<Object>();
lista.add("quack");
listaStr.add("piu-piu");
listaStr.add( new Object()); //Não aceita Object por estar a ser utilizada com a referência lista1, a qual possui limite inferior.
```

# Generics - Limitações de uso

## Algumas limitações no uso de Generics [Boyarsky&Selikoff, 2016]

- Não pode-se criar uma instância (i.e. objeto) com base em um tipo genérico, i.e., **new T()**, onde T é o nome de um tipo genérico;
- Não pode-se criar diretamente arrays de tipos genéricos, de forma similar à criação de instâncias referida anteriormente;
- Não pode-se utilizar o operador **instanceof** com um tipo genérico;
- Não pode-se utilizar diretamente um tipo primitivo em substituição a um tipo genérico, já que existem as classes *Wrappers* (i.e. Integer, Float, etc...) a representar os referidos tipos;
- Não pode-se criar atributos/variáveis estáticas com tipos genéricos.

# Collections

## O que são as famosas Collections [no bom Português Coleções]?

Segundo [Boyarsky&Selikoff, 2016] uma coleção é um grupo de objetos contidos em um único objeto. Essa definição nos remete à ideia de uma "caixinha" de objetos, onde a "caixinha" por si só é um objeto, e a organização utilizada dentro da "caixinha" especifica o tipo de estrutura de dados que está a ser usada no armazenamento interno. Portanto, a linguagem Java fornece um arcabouço (i.e. *framework*) chamado *Java Collections*, o qual é especificado no pacote *java.util*.

# Collections

## Principais interfaces do Java Collections...

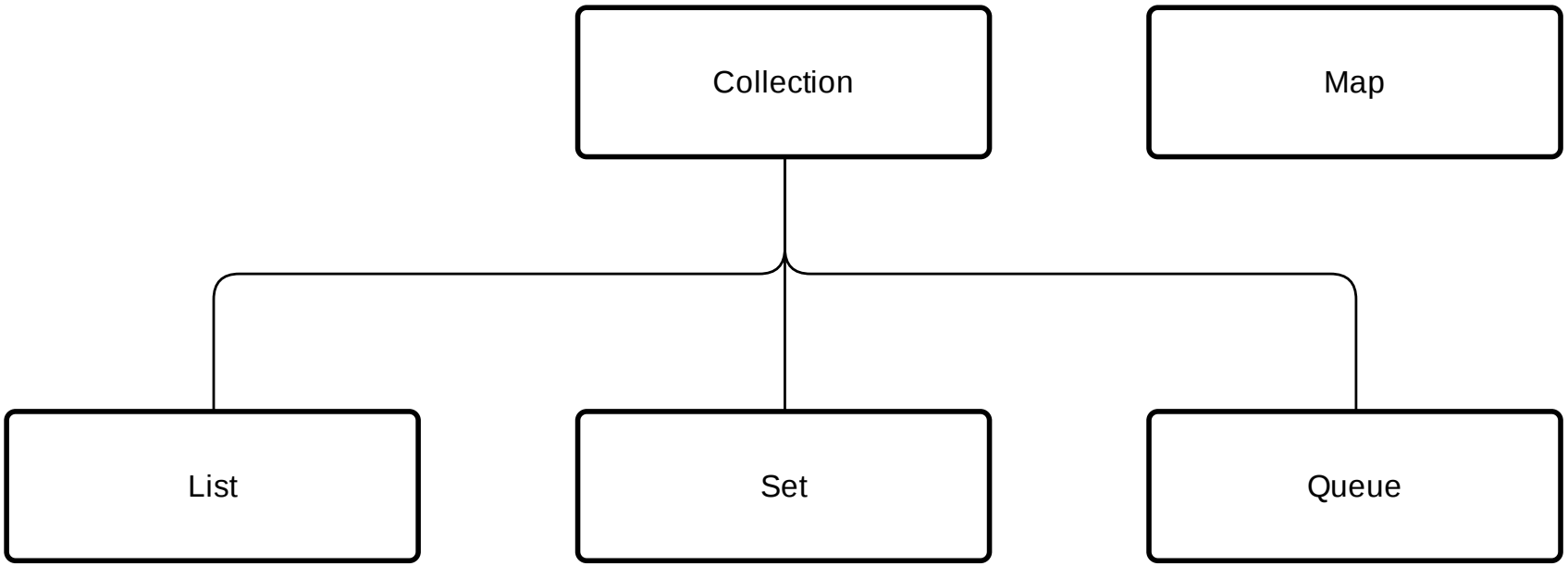
- **List**: coleção de elementos ordenados que permite o armazenamento de elementos duplicados. Os referidos elementos são acessados por um índice, o qual indica a sua ordem. Essa interface fornece uma abstração para concretização de uma estrutura de dados chamada Lista [Vocês já devem conhecer a "tipa" :-D...];
- **Set**: coleção que não permite o armazenamento de elementos duplicados, a fornecer uma abstração que representa a abstração de conjunto oriunda da Matemática;

# Collections

## Principais interfaces do Java Collections...

- **Queue**: coleção que ordena os elementos em uma ordem específica para processamento. Essa interface fornece uma abstração para concretização de uma estrutura de dados chamada Fila, cuja ordem típica é indicada pela sigla FIFO (First-In-Fisrt-Out); a ordem FIFO significa o primeiro elemento a entrar será o primeiro elemento a sair e/ou ser processado;
- **Map**: coleção a organizar os elementos em tuplas/pares de chave e valor, a não permitir chaves duplicadas. Essa interface permite a concretização de uma estrutura de dados chamada de Dicionário.

# Collections - Hierarquia das Interfaces




Hierarquia das Interfaces do *Java Collections* [Boyarsky&Selikoff, 2016], que são Interfaces Genéricas. Agora, está-se pronto a dizer: "Olá Enfermeiras!!!" :-D....

# Collections - A Interface Collection

## Principais métodos da interface Collection...

A interface *Collection* possui métodos que são comuns às interfaces *List*, *Set*, e *Queue*. Os principais são:

- **add()**: especifica assinatura de comportamento que permite inserir um novo elemento de uma coleção;
- **remove()**: especifica a assinatura de comportamento que permite remover um elemento de uma coleção;
- **isEmpty()**: especifica a assinatura de comportamento que permite verificar se uma coleção não contém elementos;
- **size()**: especifica a assinatura de comportamento que permite obter o número de elementos armazenados por uma coleção;
- **clear()**: especifica a assinatura de comportamento que permite remover todos os elementos armazenados por uma coleção;
- **contains()**: especifica a assinatura de comportamento que permite verificar se elemento faz parte de uma coleção.

# Collections - Exemplos de uso do método add() [Boyarsky&Selikoff, 2016]

```
List<String> lista = new ArrayList<>();
System.out.println(lista.add("Sparrow"));//retorna true (verdadeiro).
System.out.println(lista.add("Sparrow"));//retorna true (verdadeiro), já que permite duplicação.

Set<String> conjunto = new HashSet<>();
System.out.println(conjunto.add("Sparrow"));//retorna true (verdadeiro).
System.out.println(conjunto.add("Sparrow"));//retorna false (falso), já que não permite duplicação.
```

# Collections - Exemplos de uso do método remove() [Boyarsky&Selikoff, 2016]

```
List<String> passarinhos = new ArrayList<>();
passarinhos.add("falcão");//Lista →[falcão]
passarinhos.add("falcão");//Lista →[falcão, falcão]
System.out.println(passarinhos.remove("cardeal"));//imprime false (falso), já que a String "cardeal" não está na lista.
System.out.println(passarinhos.remove("falcão"));//imprime true (verdadeiro), a remover somente o primeiro elemento encontrado.
System.out.println(passarinhos);//Lista →[falcão]
```

# Collections - Exemplos de uso dos métodos isEmpty() e size() [Boyarsky&Selikoff, 2016]

```
List<String> passarinhos = new ArrayList<>();
System.out.println(passarinhos.isEmpty());//imprime true (verdadeiro).
System.out.println(passarinhos.size());//imprime 0 (zero).
passarinhos.add("falcão");//Lista →[falcão]
passarinhos.add("falcão");//Lista →[falcão, falcão]
System.out.println(passarinhos.isEmpty());//imprime false (falso).
System.out.println(passarinhos.size());//imprime 2 (dois).
```

# Collections - Exemplos de uso do método clear() [Boyarsky&Selikoff, 2016]

```
List<String> passarinhos = new ArrayList<>();
passarinhos.add("falcão");//Lista →[falcão]
passarinhos.add("falcão");//Lista →[falcão, falcão]
System.out.println(passarinhos.isEmpty());//imprime false (falso).
System.out.println(passarinhos.size());//imprime 2 (dois).
System.out.println(passarinhos.clear());//[ ]
System.out.println(passarinhos.isEmpty());//imprime true (verdadeiro).
System.out.println(passarinhos.size());//imprime 0 (zero).
```

# Collections - Exemplos de uso do método contains() [Boyarsky&Selikoff, 2016]

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

passarinhos.add("falcão");//Lista →[falcão]

System.out.println(passarinhos.contains("falcão"));//imprime true (verdadeiro).

System.out.println(passarinhos.contains("pintarroxo"));//imprime false (falso).
```

# Collections - Detalhe importante do funcionamento interno das Coleções

## Internamente coleções utilizam os métodos equals() e hashCode()...

Internamente as coleções utilizam os métodos *equals()* e *hashCode()* especificados na classe *java.lang.Object* para realizar comparações e armazenamento de elementos. As comparações são realizadas por meio do método *equals()*, enquanto o armazenamento de um objeto como chave em um mapa é realizado com o auxílio do método *hashCode()*. Esses métodos estão relacionados, já que dois objetos iguais devem possuir o mesmo *hashCode()*. Mais detalhes serão apresentados juntamente com a interface *Map*.

# Collections - A interface List

## Principais métodos utilizados na manipulação efetuada com a interface List [Boyarsky&Selikoff, 2016]

- **boolean add(E element)**: adiciona elemento no final da lista;
- **void add(int index, E element)**: adiciona elemento no índice informado, a mover todos os elementos, a partir do referido índice, uma posição para direita;
- **E get(int index)**: retorna o elemento presente no índice informado;
- **int indexOf(Object o)**: retorna o primeiro índice a fazer "match" com o objeto informado, ou -1 caso o referido objeto não seja encontrado;

# Collections - A interface List

## Principais métodos utilizados na manipulação efetuada com interface List [Boyarsky&Selikoff, 2016] (Continuação)

- **int lastIndexOf(Object o)**: retorna o último índice a fazer "match" com o objeto informado, ou -1 caso o referido objeto não seja encontrado;
- **E remove(int index)**: remove o elemento no índice informado, a mover todos os elementos em índices posteriores uma posição para esquerda;
- **boolean remove(E element)**: remove o elemento informado, a mover todos os elementos em índices posteriores uma posição para esquerda;
- **E set(int index, E element)**: substitui o elemento no índice informado pelo elemento passado como parâmetro.

# Collections - Iterar valores de uma Lista

**Existem diferentes formas de realizar iteração nos valores de uma lista...**

- utilizar o comando for com índices inteiros;
- utilizar o comando for com sua notação *forEach*;
- utilizar o comando while juntamente com a interface *java.util.Iterator*;
- utilizar *Streams*, que será visto mais a frente depois da apresentação do conteúdo das principais interfaces do *Java Collections*.

# Collections - Exemplo de iteração de lista com o comando for a utilizar índices inteiros

```
List<String> lFrutas = new ArrayList<>();
lFrutas.add("Banana");//Lista →[Banana]
lFrutas.add("Laranja");//Lista →[Banana, Laranja]
lFrutas.add("Limão");//Lista →[Banana, Laranja, Limão]
lFrutas.add("Manga");//Lista →[Banana, Laranja, Limão, Manga]
for(int i=0;i < lFrutas.size();i++){
  System.out.println(lFrutas.get(i));
}
```

O detalhe negativo de utilizar esse tipo de iteração é que, indiretamente, está-se a iterar de forma duplicada, a considerar que o método *get()* realiza iteração similar para encontrar o elemento a ser impresso.

# Collections - Exemplo de iteração de lista com o comando for a utilizar sua notação forEach

```
List<String> lFrutas = new ArrayList<>();
lFrutas.add("Banana");//Lista →[Banana]
lFrutas.add("Laranja");//Lista →[Banana, Laranja]
lFrutas.add("Limão");//Lista →[Banana, Laranja, Limão]
lFrutas.add("Manga");//Lista →[Banana, Laranja, Limão, Manga]
```

```
for(String fruta : lFrutas){
   System.out.println(fruta);
}
```

```
for(var fruta : lFrutas){
   System.out.println(fruta);
}
```

**A palavra reservada var pode ser utilizada a partir da versão 10.**

# Collections - Exemplo de iteração de lista com o comando while a ser utilizado juntamente com a interface Iterator

```
List<String> lFrutas = new ArrayList<>();
lFrutas.add("Banana");//Lista →[Banana]
lFrutas.add("Laranja");//Lista →[Banana, Laranja]
lFrutas.add("Limão");//Lista →[Banana, Laranja, Limão]
lFrutas.add("Manga");//Lista →[Banana, Laranja, Limão, Manga]
Iterator<String> itFrutas = lFrutas.iterator();
while(itFrutas.hasNext()){
   System.out.println(itFrutas.next());
}
```

# Bibliografia

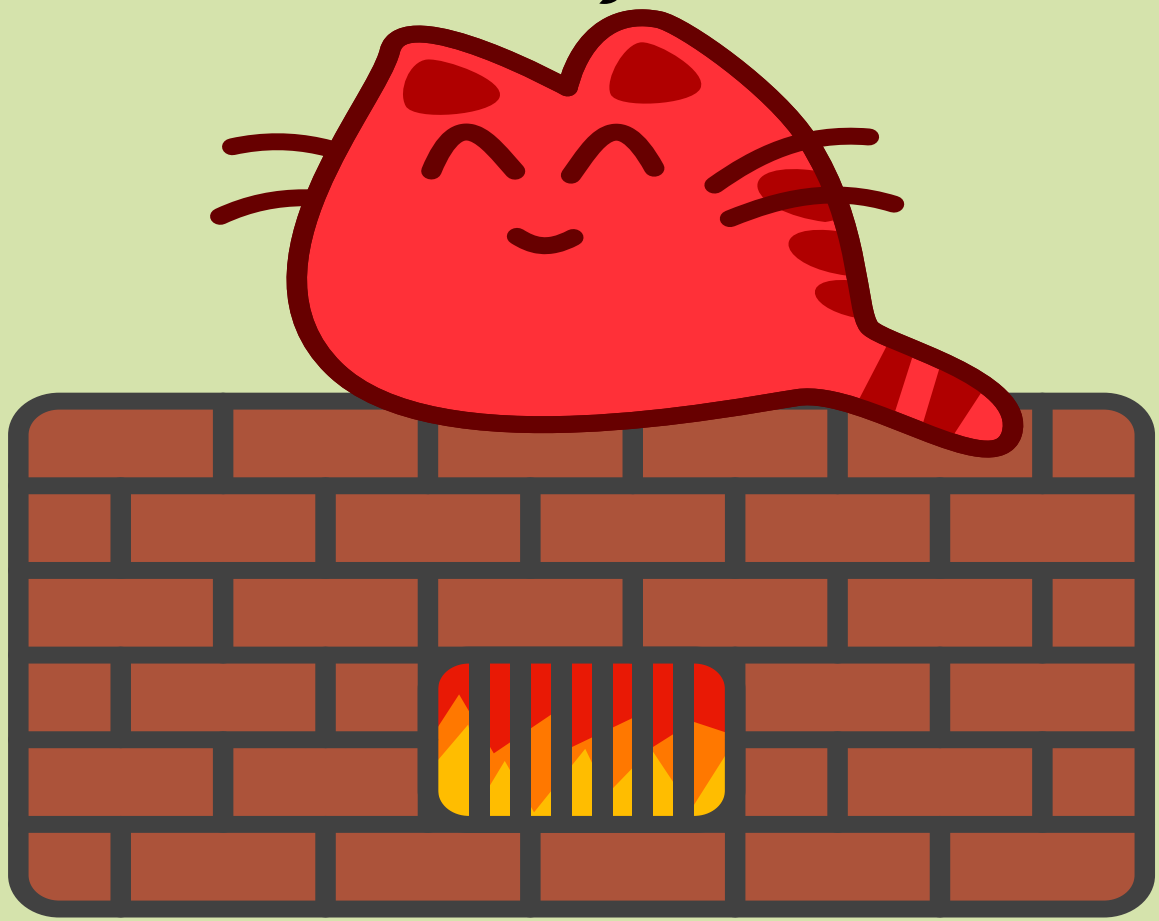

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G.; BUCKLEY, A.; SMITH, D.; BIERMAN, G. **"The Java Language Specification: Java SE 16 Edition"**. Oracle. 2021. Disponível em: `https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf`. Acesso em: 12 Jul. 2021.

BOYARSKY, J. and SELIKOFF, S. **"Oracle Certified Associate Java SE 8 Programmer I: Study Guide"**. Sybex. Indianapolis, Indiana, USA. 2015.

# Bibliografia (Continuação)

BOYARSKY, J. and SELIKOFF, S. **“Oracle Certified Associate Java SE 8 Programmer II: Study Guide”**. Sybex. Indianapolis, Indiana, USA. 2016.

*That's it folks!*



*Thank you for your attention!*