Visão Geral de Generics e Collections



Prof. Jeferson Souza, MSc. (thejefecomp) thejefecomp@neartword.com





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

Introdução

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 códigofonte, 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*



Introdução

Generics - Exemplo de Concretização de Interface Genérica

Concretizar Interface Genérica com tipo Genérico

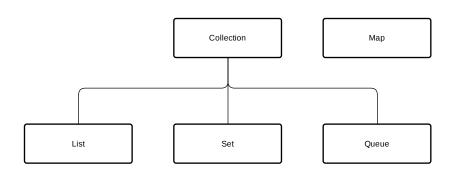
```
public class ControleImpl<T> implements Controle<T>{
    public boolean ligar(T equipamento){
      ...
    }
    public boolean desligar(T equipamento){
      ...
    }
}
```

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



Introdução

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 Definicão 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

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* "?".



Introdução

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.

O wildcard "?"...

Introdução

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.

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>();</string>
wildcard com limite superior	? extends tipo	List extends Exception lista = new ArrayList <runtimeexception>();</runtimeexception>
wildcard com limite inferior	? super tipo	List super Exception lista = new ArrayList <object>();</object>

Introdução

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);
```

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.

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 definicão 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.

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.



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.



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*.



Introdução

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;

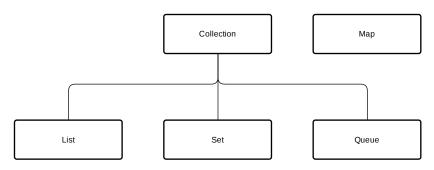


Principais interfaces do Java Collections...

- 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....



Introdução

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]

```
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<>();
```

List<String> lista = new ArrayList<>():

```
System.out.println(conjunto.add("Sparrow"));//retorna true (verdadeiro).
```

```
System.out.println(conjunto.add("Sparrow"));//retorna false (falso), já que não permite duplicação.
```

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

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

```
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 \rightarrow [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 \rightarrow[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).

Introdução

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

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(Object o): 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.lterator;
- 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> IFrutas = new ArrayList<>();
IFrutas.add("Banana");//Lista \rightarrow[Banana]
IFrutas.add("Laranja");//Lista \rightarrow[Banana, Laranja]
IFrutas.add("Limão");//Lista →[Banana, Laranja, Limão]
IFrutas.add("Manga");//Lista \rightarrow[Banana, Laranja, Limão, Manga]
for(int i=0;i < IFrutas.size();<math>i++){
  System.out.println(IFrutas.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.

Introdução

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

```
\label{linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_linear_
```

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

Collections - Concretizações mais Conhecidas da Interface List

Concretizações mais Conhecidas da Interface List

As concretizações mais conhecidas da *Interface List* são[Oracle, 2021] [Boyarsky&Selikoff, 2016]:

- LinkedList: fornece uma implementação da estrutura de dados Lista, como uma lista de elementos duplamente ligados, além de também concretizar a *Interface Queue*. Operações de adição/remoção de elementos do início e do fim da lista acontencem em tempo constante [O(1)];
- ArrayList: substitui uma antiga classe chamada Vector, a fornecer a abstração de um array em forma de lista. Os métodos size(), isEmpty(), get(), set(), iterator(), e listIterator() são executados em tempo constante [O(1)] por razão do uso de índices internos.



Collections - Diferentes Concretizações da Interface List

Detalhe interessante...

Classes tal como *java.util.Vector* e *java.util.Stack* também concretizam a *Interface List*. Porém, por serem muito antigas, e pelo fato da linguagem Java possuir alternativas mais interessantes, não devem ser utilizadas em codificações com versões mais recentes da linguagem.

Collections - A interface Set

Principais métodos utilizados na manipulação efetuada com a interface Set [Oracle, 2021]

- boolean add(E element): adiciona elemento no conjunto, caso o referido elemento já não esteja presente;
- boolean contains(Object o): verifica se um dado elemento está presente no conjunto;
- boolean isEmpty(): indica se o conjunto possui [ou não] elementos;
- boolean remove(Object o): remove o elemento informado, caso esteja presente no conjunto:
- int size(): indica o número de elementos presentes no conjunto.



Generics

Collections - Concretizações mais Conhecidas da Interface Set

Concretizações mais conhecidas da Interface Set

As concretizações mais conhecidas da Interface Set são [Oracle, 2021] [Boyarsky&Selikoff, 2016]:

- ▶ HashSet: armazena os elementos em uma tabela hash. Os métodos de add(), remove(), contains(), e size() são executados em tempo constante [O(1)];
- ► TreeSet: armazena os elementos em uma estrutura de árvore ordenada. cuja concretização é baseada na classe TreeMap, a qual utiliza a estrutura de árvore Red-Black como base de sua implementação. Os métodos add(), remove(), e contains() são executados em tempo logarítmo [O(log n)]. A classe TreeSet também concretiza a Interface NavigableSet, a qual permite navegar nos elementos de uma outra perspectiva.



Introdução

Collections - Exemplo uso da Interface Set e sua Concretização HashSet [Boyarsky&Selikoff, 2016]

```
Set < Integer > conjunto = new HashSet <> ();
conjunto.add(66);// true
conjunto.add(10);// true
conjunto.add(66);// false
conjunto.add(8);// true
for(Integer inteiro : conjunto) System.out.println(inteiro +",");
```

Collections - Exemplo uso da Interface Set e sua Concretização TreeSet [Boyarsky&Selikoff, 2016]

```
Set<Integer> conjunto = new TreeSet<>();
conjunto.add(66);// true
conjunto.add(10);// true
conjunto.add(66);// false
conjunto.add(8);// true
for(Integer inteiro : conjunto) System.out.println(inteiro + ",");
```

A diferenca para o uso da classe HashSet é que a concretização fornecida pela classe TreeSet realiza um armazenamento ordenado.



Collections - A interface Queue

Introdução

Principais métodos utilizados na manipulação efetuada com a interfaceQueue [Oracle, 2021] [Boyarsky&Selikoff, 2016]

- ▶ boolean add(E e): adiciona elemento na fila. Dispara uma exceção caso não seja possível o fazer;
- **E element()**: retorna o elemento que está na cabeça da fila (sem o remover). Dispara uma exceção caso a fila esteja vazia;
- **boolean offer(E e)**: adiciona elemento na fila, a retornar falso (false) caso não seja possível;



Collections - A interface Queue

Introdução

Principais métodos utilizados na manipulação efetuada com a interfaceQueue [Oracle, 2021] [Boyarsky&Selikoff, 2016] (Continuação)

- ► E peek(): retorna o elemento que está na cabeça da fila (sem o remover), a retornar nulo (null) caso a fila esteja vazia;
- ► E poll(): retorna e remove o elemento que está na cabeça da fila, a retornar nulo (null) caso a fila esteja vazia;
- E remove(): retorna e remove o elemento que está na cabeça da fila. Dispara uma exceção caso a fila esteja vazia.



Collections - Concretizações mais Conhecidas da Interface Queue

Concretizações mais conhecidas da Interface Queue

As concretizações mais conhecidas da *Interface Queue* são[Oracle, 2021] [Boyarsky&Selikoff, 2016]:

- LinkedList: já comentada em transparências anteriores;
- ArrayDeque: concretização da Interface Queue com abstração de array. Boa parte dos métodos é executado no que é chamado de "tempo constante amortizado" [O(m)], i.e., um tempo constante que depende do índice que o elemento ocupa dentro da estrutura de dados, o qual vai definir o valor de m.



Collections - Exemplo de uso da Interface Queue e sua Concretização ArrayDequeue [Boyarsky&Selikoff, 2016]

```
Queue < Integer > fila = new Array Deque < >();
System.out.println(fila.offer(10));// imprime true
System.out.println(fila.offer(4));// imprime true
System.out.println(fila.peek());// imprime 10
System.out.println(fila.poll());// imprime 10
System.out.println(fila.poll());// imprime 4
System.out.println(fila.peek());// imprime null
```

Collections - A interface Map

Generics

Principais métodos utilizados na manipulação efetuada com a interfaceMap [Oracle, 2021] [Boyarsky&Selikoff, 2016]

- void clear(): remove todas as chaves e valores do mapa;
- boolean containsKey(Object key): verifica se uma dada chave está presente no mapa;
- boolean containsValue(Object value): verifica se um dado valor está presente no mapa;
- V get(Object key): retorna o valor mapeado para a dada chave, ou nulo (null) se nenhum valor estiver mapeado;
- boolean isEmpty(): verifica se o mapa está vazio;



Collections - A interface Map

Principais métodos utilizados na manipulação efetuada com a interfaceMap [Oracle, 2021] [Boyarsky&Selikoff, 2016] (Continuação)

- Set<K> keySet(): retorna um conjunto com todas as chaves presentes no mapa;
- V put(K key,V value): insere a associação do dado par [chave,valor] no mapa. Caso a chave já exista no mapa, retorna o valor que já estiver mapeado; caso contrário o retorno é nulo (null);
- V remove(Object key): remove e retorna o valor mapeado para a dada chave, ou nulo (null) caso não exista nenhum mapeamento prévio;
- int size(): retorna o número de entradas, i.e. pares [chave,valor] presentes no mapa;
- Collection < V > values(): retorna uma coleção com todos os valores presentes no mapa.



Collections - Concretizações mais Conhecidas da Interface Map

Concretizações mais conhecidas da Interface Map

As concretizações mais conhecidas da *Interface Map* são[Oracle, 2021] [Boyarsky&Selikoff, 2016]:

- ► HashMap: armazena os pares [chave,valor] em uma tabela hash, a qual utiliza o método hashCode() durante o processo de armazenamento. Os métodos get() e put() oferecem execuções em tempo constante [O(1)], a assumir que a função de hash distribui bem os elementos na estrutura de buckets (baldes) utilizada pelo armazenamento interno;
- ► TreeMap: armazena os pares [chave,valor] em uma estrutura de árvore ordenada Red-Black como base de sua implementação. Os métodos containsKey(), get(), put(), e remove() oferecem execuções em tempo logarítimico [O(log n)].



Collections - Exemplo de uso da Interface Map e sua Concretização HashMap [Boyarsky&Selikoff, 2016]

```
Map<String> mapa = new HashMap<>();
mapa.put("coala", "bambu");
mapa.put("leão", "carne");
mapa.put("girafa", "folha");
String comida = mapa.get("coala");// bambu
for(String chave : mapa.keySet()) System.out.println(chave +",");
```

Collections - A importância dos métodos hashCode() e equals() para as Coleções

Contrato do método hashCode()

O método hashCode() possui o seguinte contrato [Oracle, 2021]:

- Durante uma mesma execução de um dado programa, diferentes invocações do método hashCode(), sobre o mesmo objeto, devem retornar o mesmo valor;
- Caso dois objetos sejam considerados iguais, com base na comparação realizada pelo método equals(), a invocação do método hashCode() sobre os dois objetos deve retornar o mesmo valor;
- Não é obrigatório que dois objetos considerados diferentes, com base na comparação realizada pelo método equals(), possuam valores retornados pela invocação do método hashCode() diferentes. Entretanto, ao produzir valores diferentes tem-se uma melhoria no armazenamento e desempenho das tabelas hash utilizadas nos mapas.



Collections - A importância dos métodos hashCode() e equals() para as Coleções

Contrato do método equals()

O método **equals()** possui o seguinte contrato (i.e. relação de equivalência para referências não-nulas) [Oracle, 2021]:

- ► Reflexivo: para uma referência não-nula x, x.equals(x) deve retornar true (verdadeiro);
- Simétrico: para duas referências não-nulas x e y iguais, x.equals(y) e y.equals(x) devem retornar true (verdadeiro);
- ► Transitivo: para três referências não-nulas x, y, e z, se x.equals(y) retornar true (verdadeiro), e y.equals(z) também retornar true (verdadeiro), então x.equals(z) deve retornar true (verdadeiro);
- Consistente: para duas referências não-nulas x e y, múltiplas invocações x.equals(y) devem retornar o mesmo valor;
- Para uma referências não-nula x, x.equals(null) deve retornar false (falso).



Introdução

Collections - A classe java.util.Collections

Prazer, chamo-me java.util.Collections, o canivete suíço das coleções...

A classe java.util.Collections possui métodos estáticos para a manipulação de coleções, a incluir métodos de ordenação e busca de elementos em uma lista. A seguir são mostrados alguns exemplos de utilização.

Collections - Exemplo de uso da classe java.util.Collections

```
para ordenar uma lista na ordem natural (crescente) dos elementos. List<Integer> INumeros = new ArrayList<>();
```

```
INumeros.add(55);//Lista \rightarrow[55]
INumeros.add(6);//Lista \rightarrow[55,6]
INumeros.add(2);//Lista \rightarrow[55,6,2]
```

INumeros.add(34);//Lista \rightarrow [55,6,2,34]

INumeros.add(40);//Lista \rightarrow [55,6,2,34,40]

Collections.sort(INumeros);//Lista \rightarrow [2,6,34,40,55]

O algoritmo de ordenação utilizado é uma adaptação do famoso algoritmo de ordenação *mergesort()*.



Collections - Exemplo de uso da classe java.util.Collections

```
para retornar a ordem reversa de elementos em uma lista.
List<Integer> INumeros = new ArrayList<>();
```

```
INumeros.add(55);//Lista \rightarrow[55]
```

INumeros.add(6);//Lista
$$\rightarrow$$
[55,6]

INumeros.add(2);//Lista
$$\rightarrow$$
[55,6,2]

INumeros.add(34);//Lista
$$\rightarrow$$
[55,6,2,34]

INumeros.add(40);//Lista
$$\rightarrow$$
[55,6,2,34,40]

Collections.reverse(INumeros);//Lista
$$\rightarrow$$
[40,34,2,6,55]

Collections - Exemplo de uso da classe java.util.Collections

```
\label{linear} \begin{tabular}{ll} List<&lnteger>&lNumeros = new ArrayList<>>();\\ &lNumeros.add(55);//Lista $\rightarrow$[55]\\ &lNumeros.add(6);//Lista $\rightarrow$[55,6]\\ &lNumeros.add(2);//Lista $\rightarrow$[55,6,2]\\ &lNumeros.add(34);//Lista $\rightarrow$[55,6,2,34]\\ &lNumeros.add(40);//Lista $\rightarrow$[55,6,2,34,40]\\ &Collections.sort(lNumeros);//Lista $\rightarrow$[2,6,34,40,55]\\ \end{tabular}
```

System.out.println(Collections.binarySearch(INumeros,34));// imprime 2

PS: para realizar busca binária de elementos a lista precisa estar ordenada em ordem crescente.



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.



ORACLE, INC. "Java Platform, Standard Edition & Java Development Kit Version 16 API Specification". Oracle. 2021. Disponível em: https://docs.oracle.com/en/java/javase/16/docs/api/index.html. Acesso em: 26 Jul. 2021.

Bibliografia (Continuação)



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



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

