

Principais APIs Java, Threads, Anotações e Genéricos

Prof. Emílio Dias
emiliodias@gmail.com
<http://www.github.com/emiliodias>

Conteúdo da disciplina

- Java API's
- Collections
- API's Java 8
- API's I/O
- Genéricos
- Anotações
- Threads

Avaliação

- 100 pontos divididos em:
 - 10 pontos por lista de exercício (8 listas)
 - 20 pontos código final

API Java

O pacote **java** e **javax**

O pacote **java** é o principal pacote da plataforma, dentro deste pacote encontram-se outros 13 pacotes que possuem boa parte das implementações que utilizamos.

As outras funcionalidades básicas se encontram sob o pacote **javax**.

API Java

Conteúdo do pacote java:

Pacote	Descrição
applet	Provê as classes necessárias para criar um applet e as classes que um applet usa para se comunicar com seu contexto.
awt	Contém classes e interfaces utilizadas para desenhar gráficos e imagens e construir GUIs.
beans	Contém classes relacionadas ao desenvolvimento de componentes beans baseados na arquitetura denominada como JavaBean.
io	Provê entrada e saída para o sistema através de fluxo, serialização e arquivos de sistema.
lang	Provê classes que são fundamentais ao desígnio de Java enquanto linguagem de programação.
math	Provê classes para executar aritmética de inteiros de precisão arbitrária e aritmética decimal de precisão.
net	Provê classes para implementação de aplicações de redes.
nio	Define “buffers”, que são recipientes para dados e proveem uma prévia dos outros pacotes NIO.
rmi	Provê classes e interfaces para implementar aplicativos que utilizem invocação remota de métodos (RMI – Remot Method Invocation.)
security	Provê classes e interfaces para implementar procedimentos de segurança de informações.
sql	Provê classes e interfaces para acessar a processar dados armazenados em uma fonte de dados, normalmente um banco de dados relacional.
text	Provê classes e interfaces para controlar texto, datas, números e mensagens de modo independente de idiomas naturais.
util	Contém a estrutura de coleções, modelo de eventos, facilidades com data e hora, internacionalização e classes de utilidades diversas.

API Java

Parte do conteúdo do pacote javax:

Pacote	Descrição
accessibility	Define um contrato entre componentes de interface do usuário e uma tecnologia que provê acesso a esses componentes.
imageio	Pacote principal de entrada e saída de imagem.
naming	Contém classes e interfaces para nomear acesso a serviços.
print	Contém as classes e interfaces principais para o serviço de impressão do Java.
rmi	Contém classes e interfaces adicionais para a implementação de invocação remota de métodos.
security	Provê uma estrutura para autenticação e autorização através de certificados e chaves públicas.
sound	Provê classes e interfaces para capturar, processamento e reprodução de áudio.
sql	Provê acesso a fonte de dados do lado do servidor.
swing	Provê um conjunto de componentes “leves” para a construção de GUIs que funcionam do mesmo modo em todas as plataformas.
transaction	Contém classes de exceção lançadas pelo Object Request Broker (ORB).
xml	Provê classes que permitem o processamento de documentos XML.

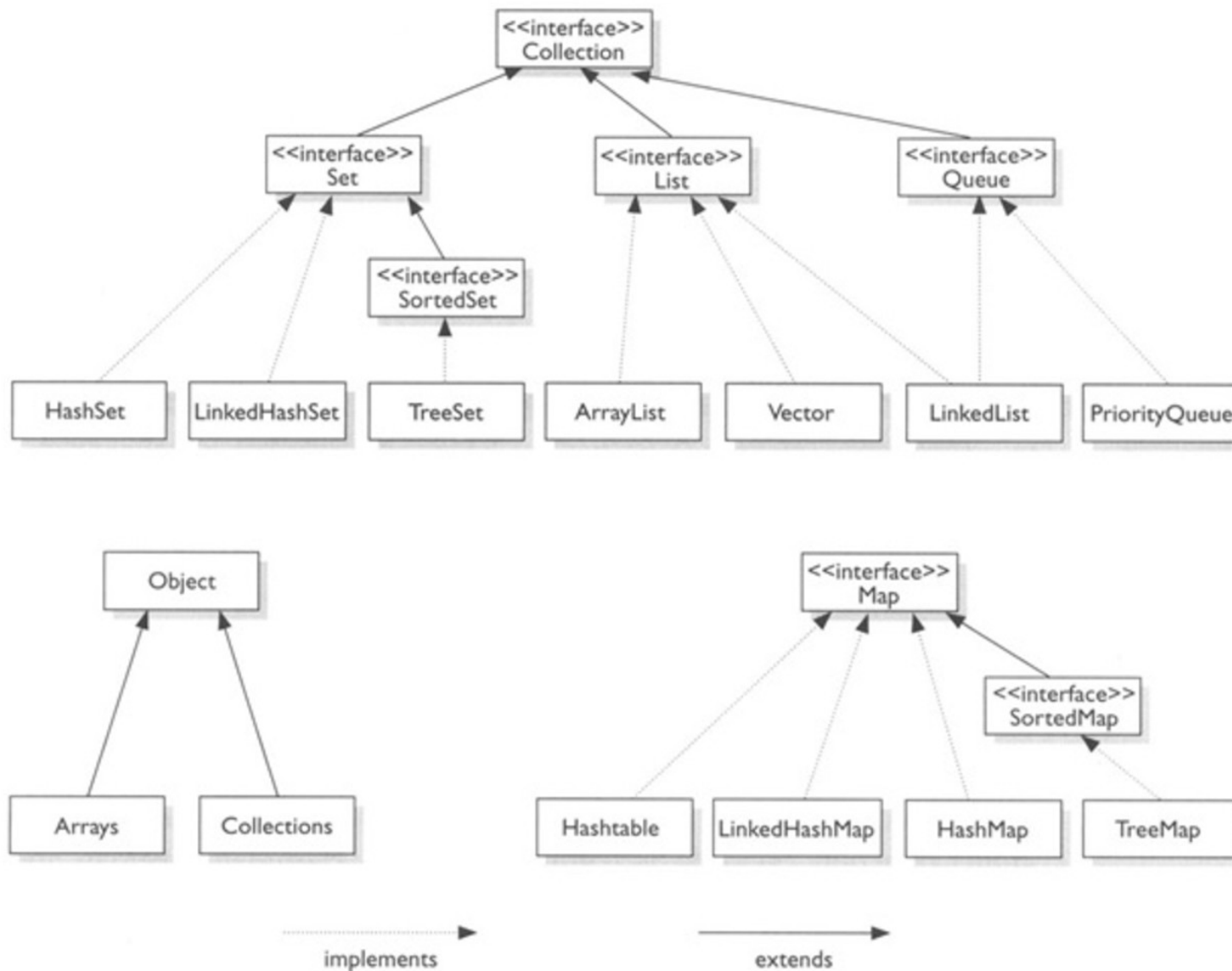
Collections

Uma API que nos oferece as principais estruturas de dados, como Listas, Filas, Conjuntos, Mapas de valores e etc.

Collections

- **Set** – define uma coleção que não contém valores duplicados
- **Queue** – define uma coleção que representa uma fila
- **List** – define uma coleção ordenada que pode conter elementos duplicados

Collections



Collections

Coleções podem ser:

- Ordenadas
- Não ordenadas
- Sorteadas
- Não sorteadas

Collections

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
Hashtable	x			No	No
TreeMap	x			Sorted	By <i>natural order</i> or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By <i>natural order</i> or custom comparison rules
LinkedHashSet		x		By insertion order	No
ArrayList			x	By index	No
Vector			x	By index	No
LinkedList			x	By index	No
PriorityQueue				Sorted	By to-do order

Fonte: SCJP Sun Certified Programmer for Java 6 Study Guide

java.util.List

- Uma das principais interfaces da API de Collections
- Elementos ordenados
- Elementos duplicados
- Inserção, busca e remoção ordenadas
- Possui diversas implementações, sendo a principal ArrayList

java.util.ArrayList

Criando um ArrayList:

```
ArrayList lista = new ArrayList();
```

ATENÇÃO!!!

Não se faz necessário
informar o
tamanho da lista
quando a declaramos.

ArrayList é uma
estrutura dinâmica!!!

Utilizando a interface List:

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

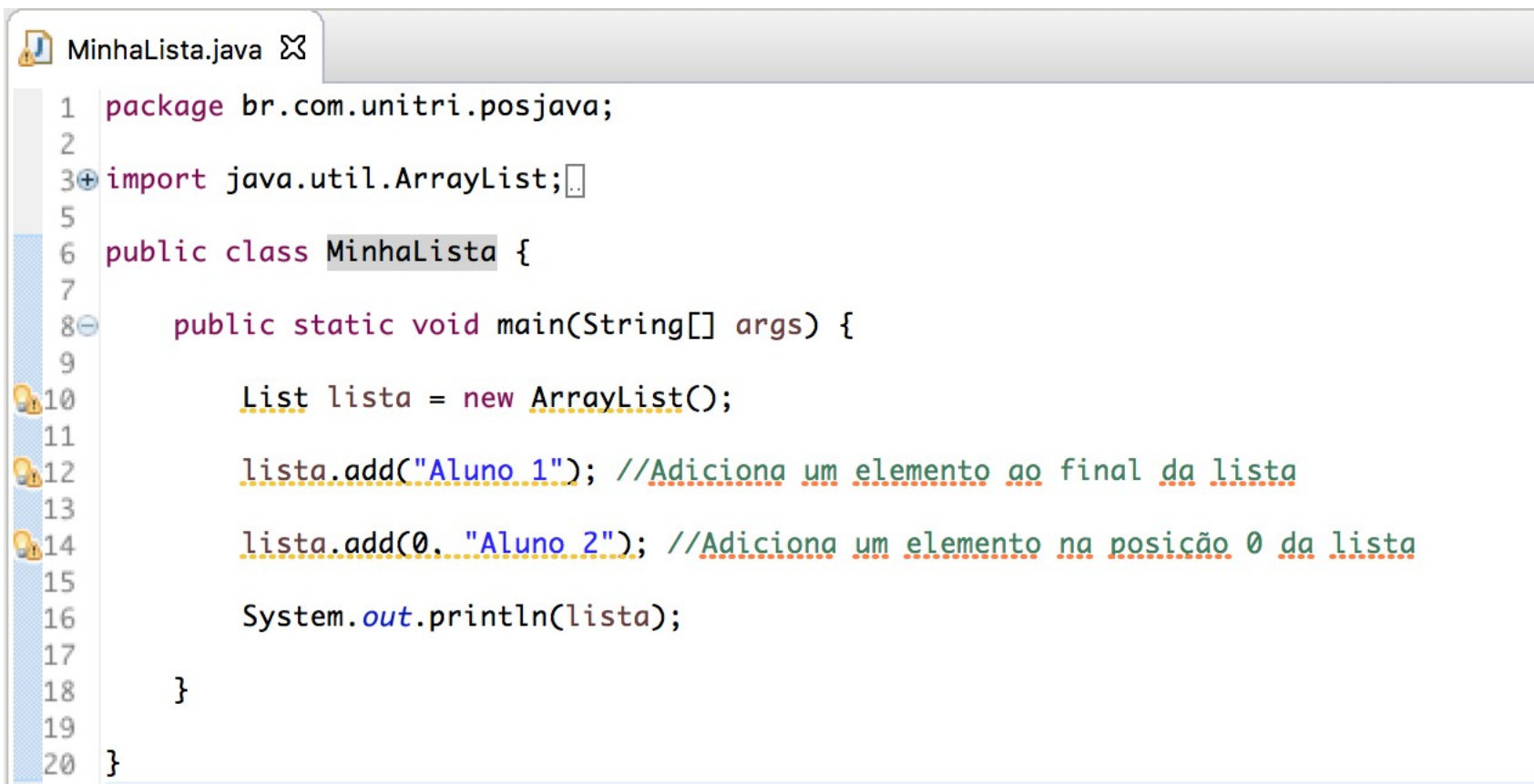
java.util.ArrayList

Adicionando elementos:

```
List lista = new ArrayList();  
lista.add("Aluno 1");  
lista.add("Aluno 2");  
lista.add("Aluno 3");
```

java.util.ArrayList

Adicionando elementos:



```
MinhaLista.java ✖
1 package br.com.unitri.posjava;
2
3 import java.util.ArrayList;
4
5
6 public class MinhaLista {
7
8     public static void main(String[] args) {
9
10         List lista = new ArrayList();
11
12         lista.add("Aluno 1"); //Adiciona um elemento ao final da lista
13
14         lista.add(0, "Aluno 2"); //Adiciona um elemento na posição 0 da lista
15
16         System.out.println(lista);
17
18     }
19
20 }
```

java.util.ArrayList

Buscando elementos:

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

```
lista.add("Aluno 1");
```

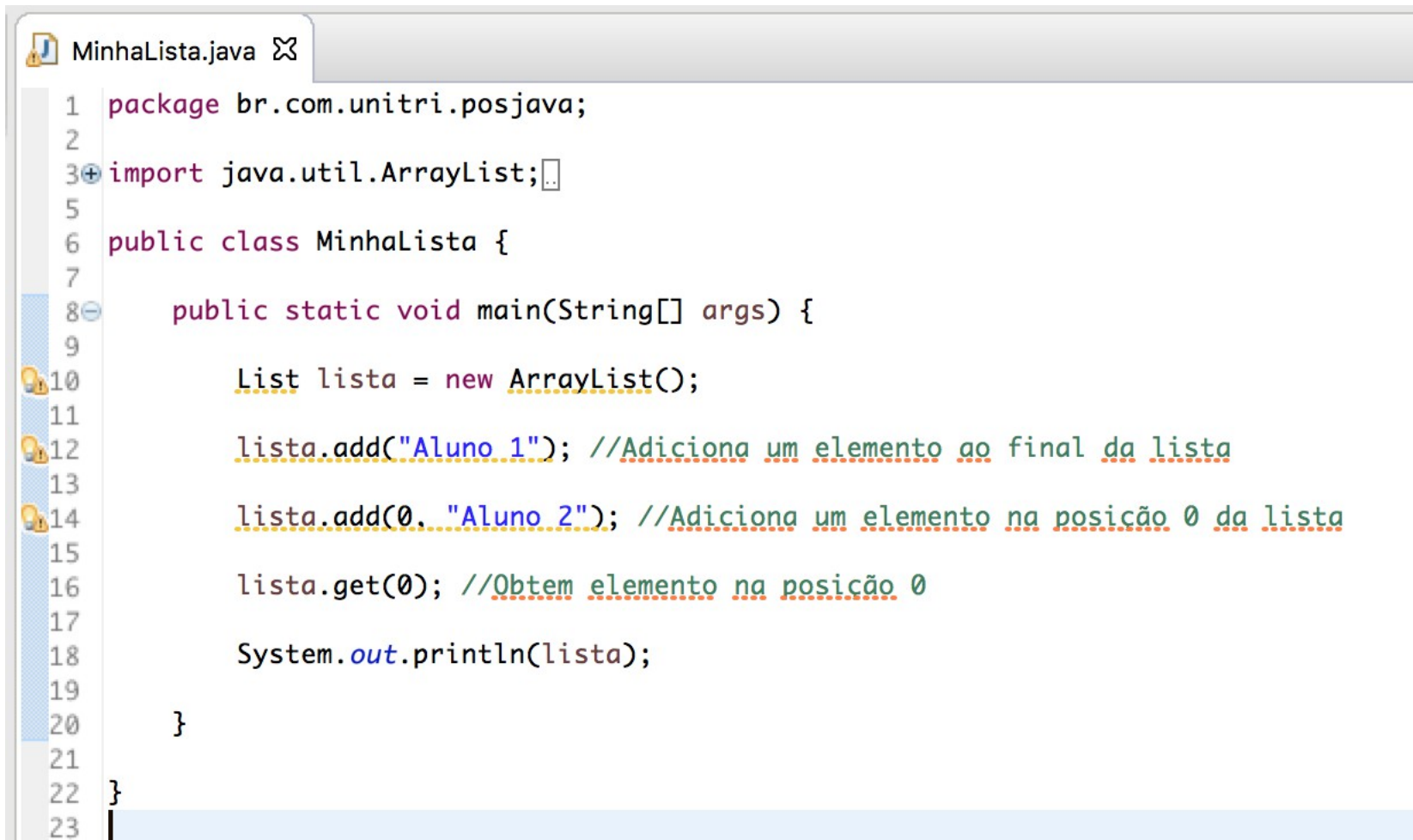
```
lista.add("Aluno 2");
```

```
lista.add("Aluno 3");
```

```
lista.get(0);
```


java.util.ArrayList

Buscando elementos:



```
1 package br.com.unitri.posjava;
2
3 import java.util.ArrayList;
4
5
6 public class MinhaLista {
7
8     public static void main(String[] args) {
9
10         List lista = new ArrayList();
11
12         lista.add("Aluno 1"); //Adiciona um elemento ao final da lista
13
14         lista.add(0, "Aluno 2"); //Adiciona um elemento na posição 0 da lista
15
16         lista.get(0); //Obtem elemento na posição 0
17
18         System.out.println(lista);
19
20     }
21
22 }
23
```

java.util.ArrayList

Removendo elementos:

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

```
lista.add("Aluno 1");
```

```
lista.add("Aluno 2");
```

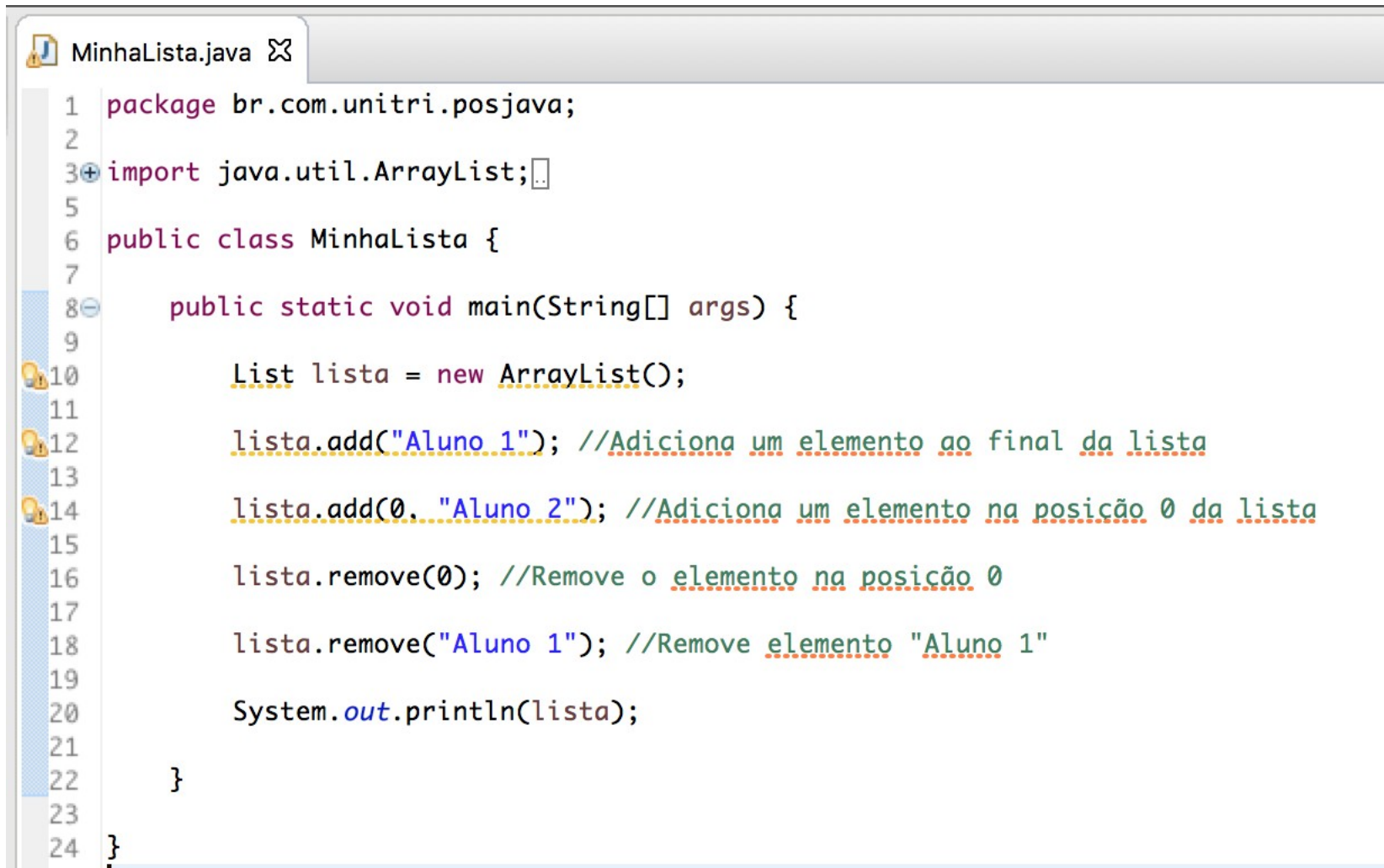
```
lista.add("Aluno 3");
```

```
lista.remove(0);
```

```
lista.remove("Aluno 1");
```

java.util.ArrayList

Removendo elementos:



```
MinhaLista.java
1 package br.com.unitri.posjava;
2
3 import java.util.ArrayList;
4
5
6 public class MinhaLista {
7
8     public static void main(String[] args) {
9
10         List lista = new ArrayList();
11
12         lista.add("Aluno 1"); //Adiciona um elemento ao final da lista
13
14         lista.add(0, "Aluno 2"); //Adiciona um elemento na posição 0 da lista
15
16         lista.remove(0); //Remove o elemento na posição 0
17
18         lista.remove("Aluno 1"); //Remove elemento "Aluno 1"
19
20         System.out.println(lista);
21
22     }
23
24 }
```

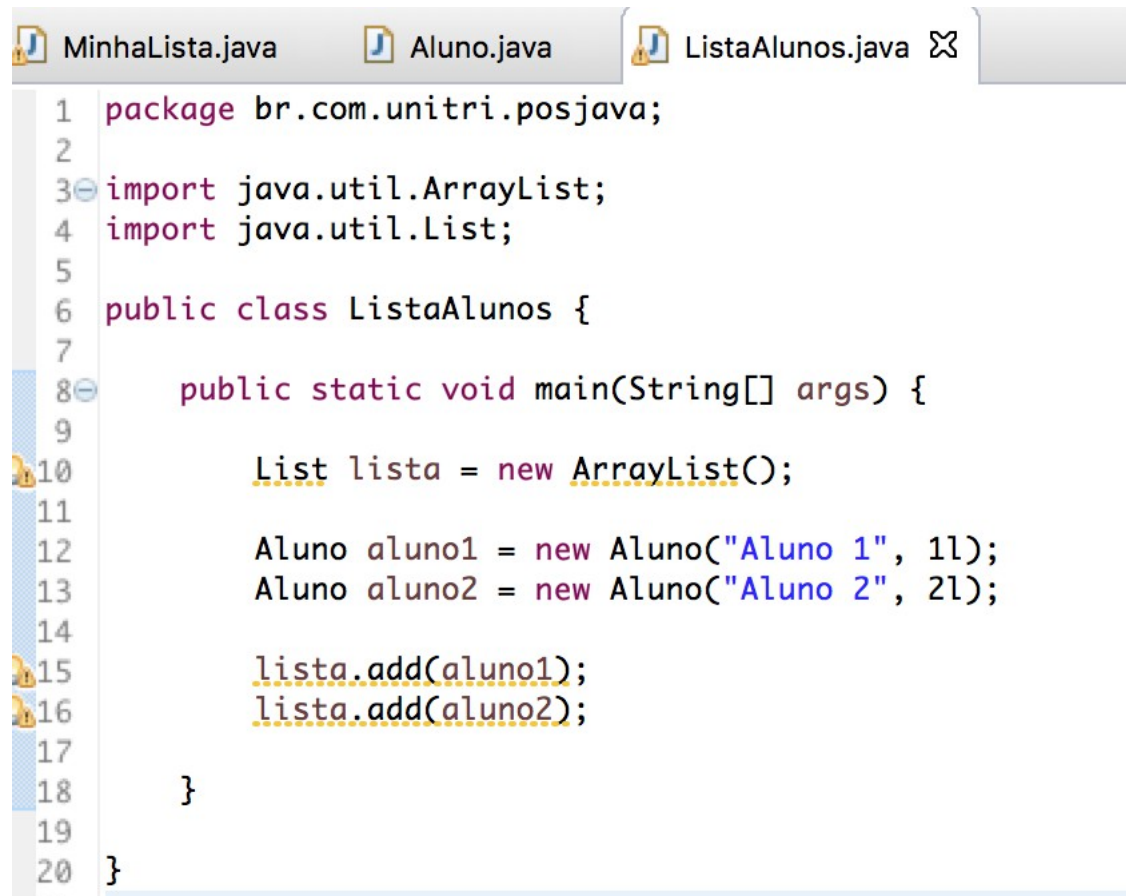
java.util.ArrayList

Outros métodos de List:

- **lista.isEmpty()** - Verifica se a lista está vazia
- **lista.size()** - Retorna o tamanho da lista
- **lista.clear()** - Remove todos os elementos da lista
- **lista.contains(value)** — Verifica se a lista contém um determinado elemento
- **lista.indexOf(value)** — Retorna a posição de um determinado elemento
- **lista.addAll(novaLista)** - Adiciona uma outra lista ao final da primeira
- **lista.sort(Ordenador)** — Ordena a lista

java.util.ArrayList

Manipulando lista de objetos



```
MinhaLista.java  Aluno.java  ListaAlunos.java ✕  
1 package br.com.unitri.posjava;  
2  
3 import java.util.ArrayList;  
4 import java.util.List;  
5  
6 public class ListaAlunos {  
7  
8     public static void main(String[] args) {  
9  
10         List lista = new ArrayList();  
11  
12         Aluno aluno1 = new Aluno("Aluno 1", 11);  
13         Aluno aluno2 = new Aluno("Aluno 2", 21);  
14  
15         lista.add(aluno1);  
16         lista.add(aluno2);  
17  
18     }  
19  
20 }
```

java.util.ArrayList

Iterando sobre uma lista de objetos

```
MinhaLista.java  Aluno.java  ListaAlunos.java  ✕
1  package br.com.unitri.posjava;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class ListaAlunos {
7
8      public static void main(String[] args) {
9
10         List lista = new ArrayList();
11
12         Aluno aluno1 = new Aluno("Aluno 1", 11);
13         Aluno aluno2 = new Aluno("Aluno 2", 21);
14
15         lista.add(aluno1);
16         lista.add(aluno2);
17
18         /**
19          * Iterando sobre a lista utilizando
20          * o método "get". Perceba que na linha 25
21          * é preciso a utilização de um casting para
22          * uma correta atribuição.
23          */
24
25         for(int i=0; i < lista.size(); i++) {
26
27             Aluno a = (Aluno) lista.get(i);
28
29             System.out.println(a.getNome());
30
31         }
32     }
33 }
34
35 }
36
```

java.util.ArrayList

Manipulando lista de objetos “tipadas”

```
MinhaLista.java  Aluno.java  ListaAlunos.java ✕
1  package br.com.unitri.posjava;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class ListaAlunos {
7
8      public static void main(String[] args) {
9
10         //Utilizando uma tipagem para a declaração da lista
11         List<Aluno> lista = new ArrayList<Aluno>();
12
13         Aluno aluno1 = new Aluno("Aluno 1", 11);
14         Aluno aluno2 = new Aluno("Aluno 2", 21);
15
16         lista.add(aluno1);
17         lista.add(aluno2);
18
19
20         //Versão "foreach" do FOR para iteração da lista de objetos
21         for(Aluno a : lista) {
22
23             System.out.println(a.getNome());
24
25         }
26
27     }
28
29 }
```

Definindo que a lista deve armazenar elementos do tipo “Aluno”.

java.util.ArrayList

Manipulando lista de objetos “tipadas”

```
MinhaLista.java  Aluno.java  ListaAlunos.java ✕
1  package br.com.unitri.posjava;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class ListaAlunos {
7
8      public static void main(String[] args) {
9
10         //Utilizando uma tipagem para a declaração da lista
11         List<Aluno> lista = new ArrayList<Aluno>();
12
13         Aluno aluno1 = new Aluno("Aluno 1", 11);
14         Aluno aluno2 = new Aluno("Aluno 2", 21);
15
16         lista.add(aluno1);
17         lista.add(aluno2);
18
19
20         //Versão "foreach" do FOR para iteração da lista de objetos
21         for(Aluno a : lista) {
22
23             System.out.println(a.getNome());
24
25         }
26
27     }
28
29 }
```

Definindo que a lista deve armazenar elementos do tipo “Aluno”.

java.util.ArrayList

Ordenando elementos de uma lista

```
lista.sort(new Comparator<Aluno>() {  
  
    @Override  
    public int compare(Aluno o1, Aluno o2) {  
  
        return o1.getNome().compareTo(o2.getNome());  
    }  
  
});
```

java.util.ArrayList

MinhaLista.java

Aluno.java

ListaAlunos.java

```
1 package br.com.unitri.posjava;
2
3 import java.util.ArrayList;
4 import java.util.Collections;
5 import java.util.Comparator;
6 import java.util.List;
7
8 public class ListaAlunos {
9
10     public static void main
11
12         //Utilizando uma ti
13         List<Aluno> lista =
14
15         Aluno aluno1 = new
16         Aluno aluno2 = new
17
18         lista.add(aluno1);
19         lista.add(aluno2);
20
21
22         lista.sort(new Comp
23
24         @Override
25         public int comp
26
27         return o1.g
28
29     }
30
31 }
32
33 Collections.sort(li
34
35 System.out.println(
36
37
38 }
```

int java.lang.String.compareTo(String anotherString)

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the [equals\(Object\)](#) method would return true.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let *k* be the smallest such index; then the string whose character at position *k* has the smaller value, as determined by using the < operator, lexicographically precedes the other string. In this case, compareTo returns the difference of the two character values at position *k* in the two string -- that is, the value:

this.charAt(k)-anotherString.charAt(k)

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, compareTo returns the difference of the lengths of the strings -- that is, the value:

this.length()-anotherString.length()

Specified by: [compareTo\(...\)](#) in [Comparable](#)

Parameters:
anotherString the String to be compared.

Returns:
the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

java.util.ArrayList

Ordenando elementos com a classe Collections:

`Collections.sort(lista);`

```
1 package br.com.unitri.posjava;
2
3 public class Aluno implements Comparable<Aluno>{
4
5     private String nome;
6
7     private Long matricula;
8
9     public Aluno() {}
10
11     @Override
12     public int compareTo(Aluno o) {
13
14         return this.nome.compareTo(o.getNome());
15     }
16 }
```

java.util.ArrayList

Exceções: Algumas exceções do tipo **RuntimeException** podem ser lançadas quando uma operação ilegal é executada. Exemplo:



The screenshot shows an IDE with two panes. The left pane displays the source code of a Java application named 'ListaAlunos.java'. The code defines a package, imports necessary classes, and creates a list of two 'Aluno' objects. It then attempts to retrieve an element at index 10, which is out of bounds. The right pane shows the console output, which reports a 'java.lang.IndexOutOfBoundsException' with the message 'Index: 10, Size: 2'. The stack trace indicates the exception was thrown during a range check and a get operation on the ArrayList.

```
1 package br.com.unitri.posjava;
2
3 import java.util.ArrayList;
4 import java.util.Collections;
5 import java.util.Comparator;
6 import java.util.List;
7
8 public class ListaAlunos {
9
10     public static void main(String[] args) {
11
12         //Utilizando uma tipagem para a declaração
13         List<Aluno> lista = new ArrayList<Aluno>();
14
15         Aluno aluno1 = new Aluno("BA aluno 1", 11);
16         Aluno aluno2 = new Aluno("Aluno 2", 21);
17
18         lista.add(aluno1);
19         lista.add(aluno2);
20
21         System.out.println(lista.get(10));
22
23     }
24
25 }
26 }
```

```
<terminated> ListaAlunos [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 10, Size: 2
    at java.util.ArrayList.rangeCheck(ArrayList.java:653)
    at java.util.ArrayList.get(ArrayList.java:429)
    at br.com.unitri.posjava.ListaAlunos.main(ListaAlunos.java:21)
```

java.util.ArrayList

Opte pelo uso da interface ao invés de utilizar a declaração do tipo concreto, desta maneira, seu código fica mais abstrato e com um grau menor de acoplamento. Exemplo:

```
public List<Aluno> listarAlunos() {  
    return alunos;  
}  
  
public void salvarAlunos(List<Aluno> alunos) {  
  
}
```

Perceba que o retorno e argumento utilizados, são declarados com a interface **List**, e não o tipo concreto **ArrayList**.

java.util.Collections

Além do método **sort**, a classe **Collections** possui uma série de outros métodos que podem auxiliar o seu trabalho. Sendo assim sugere-se:

- Com o auxílio de sua IDE, passe um tempo analisando os métodos disponibilizados pelas classes **ArrayList** e **Collections**.

java.util.Set

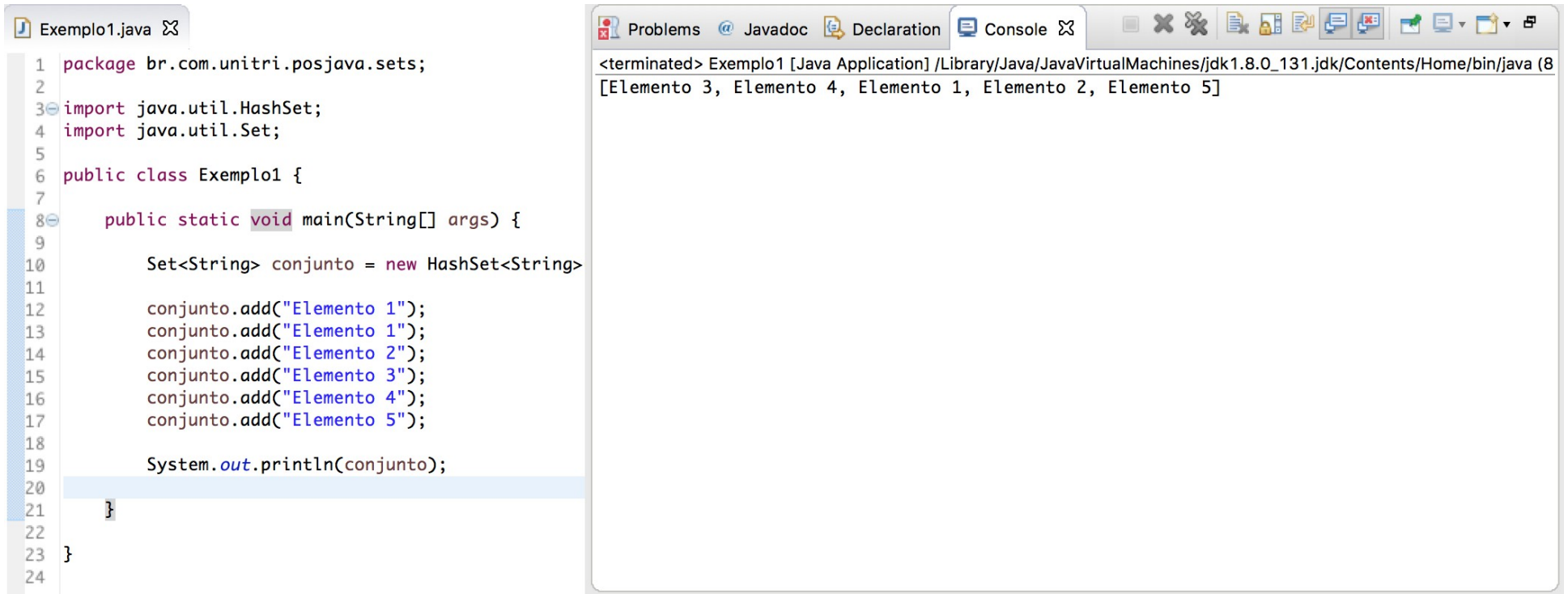
As listas são estruturas de dados que nos permite adicionar dados de forma duplicada, porém, em alguns casos é importante trabalhar com elementos que não se repitam.

As implementações de java.util.Set nos auxilia na implementação de códigos onde elementos não devem ser repetir.

Uma diferença importante entre **List** e **Set**, é que os elementos armazenados em **Set** não possuem garantias que serão retornados na mesma ordem. Apesar disso, as implementações **TreeSet** e **LinkedHashSet** possuem mecanismos para tratar esta ordenação.

Principais implementações: **HashSet**, **LinkedHashSet** e **TreeSet**

java.util.HashSet



The screenshot displays an IDE with two panels. The left panel shows a Java file named 'Exemplo1.java' with the following code:

```
1 package br.com.unitri.posjava.sets;
2
3 import java.util.HashSet;
4 import java.util.Set;
5
6 public class Exemplo1 {
7
8     public static void main(String[] args) {
9
10         Set<String> conjunto = new HashSet<String>
11
12         conjunto.add("Elemento 1");
13         conjunto.add("Elemento 1");
14         conjunto.add("Elemento 2");
15         conjunto.add("Elemento 3");
16         conjunto.add("Elemento 4");
17         conjunto.add("Elemento 5");
18
19         System.out.println(conjunto);
20
21     }
22 }
23
24
```

The right panel shows the 'Console' tab with the output of the program:

```
<terminated> Exemplo1 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (8
[Elemento 3, Elemento 4, Elemento 1, Elemento 2, Elemento 5]
```

- Elemento duplicado “Elemento 1” não inserido;
- A ordem que os elementos são obtidos é diferente da ordem na qual foram inseridos.

java.util.HashSet

Iteração:

```
1 package br.com.unitri.posjava.sets;
2
3 import java.util.HashSet;
4 import java.util.Iterator;
5 import java.util.Set;
6
7 public class Exemplo1 {
8
9     public static void main(String[] args) {
10
11         Set<String> conjunto = new HashSet<String>();
12
13         conjunto.add("Elemento 1");
14         conjunto.add("Elemento 1");
15         conjunto.add("Elemento 2");
16         conjunto.add("Elemento 3");
17         conjunto.add("Elemento 4");
18         conjunto.add("Elemento 5");
19
20         Iterator<String> i = conjunto.iterator();
21
22         while(i.hasNext()) {
23
24             System.out.println(i.next());
25
26         }
27
28     }
29 }
30
31 }
```

ATENÇÃO!!!

Uso da classe Iterator
para obter os
elementos.

- A interface Set, diferente da interface List, não possui uma forma de acessar os seus elementos por um índice.
- Além do uso do Iterator, é possível percorrer os elementos com o “foreach”

java.util.Set

ORDEM DE UM SET

Seria possível usar uma outra implementação de conjuntos, como um `TreeSet`, que insere os elementos de tal forma que, quando forem percorridos, eles apareçam em uma ordem definida pelo método de comparação entre seus elementos. Esse método é definido pela interface `java.lang.Comparable`. Ou, ainda, pode se passar um `Comparator` para seu construtor.

Já o `LinkedHashSet` mantém a ordem de inserção dos elementos.

Fonte: Apostila Caelum

java.util.Map

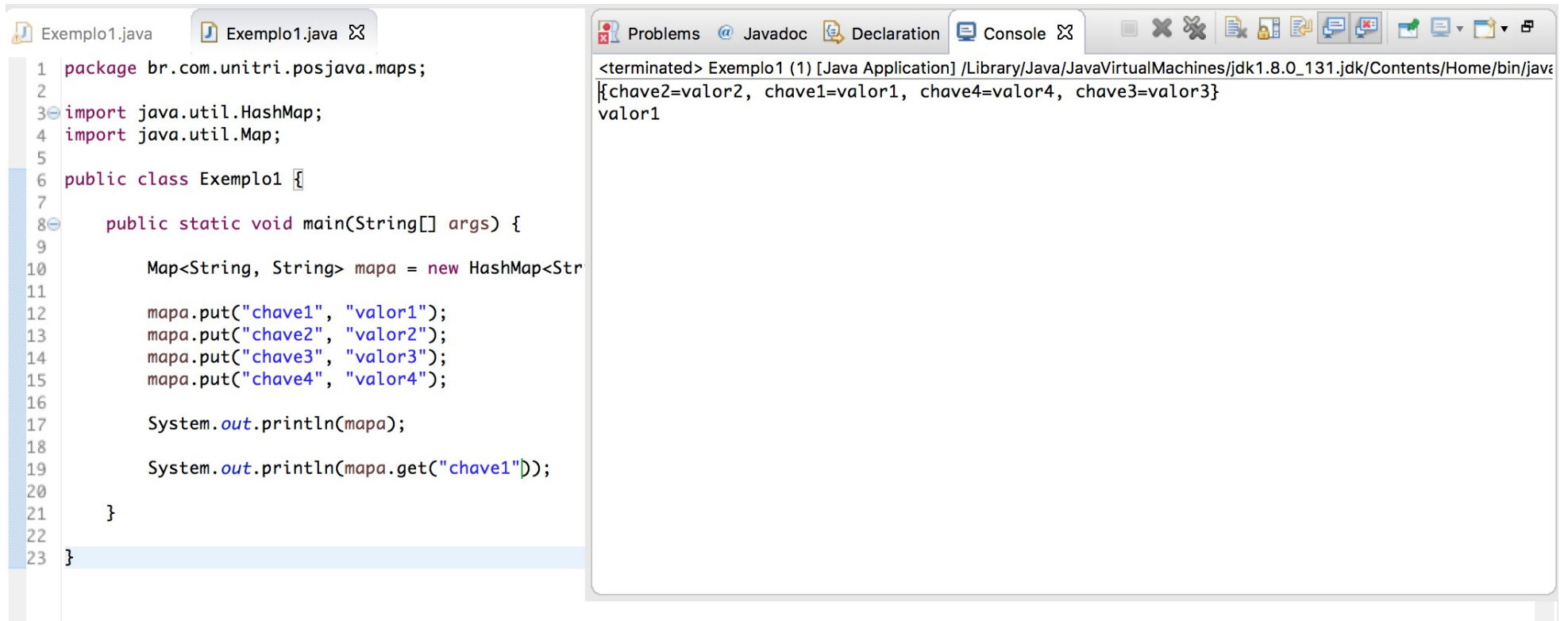
A interface Map permite que elementos sejam inseridos e recuperados baseados no modelo <CHAVE,VALOR>.

Desta forma, podemos armazenar elementos semelhantes baseados em um índice e assim acessa-los de forma mais eficiente.

Um mapa é muito usado para “indexar” objetos de acordo com determinado critério, para podermos buscar esse objetos rapidamente. Um mapa costuma aparecer juntamente com outras coleções, para poder realizar essas buscas!

Por possuir comportamentos um pouco diferentes, as implementações de Map não fazem parte das implementações de Collections.

java.util.HashMap



The screenshot displays an IDE with two panels. The left panel shows the source code for `Exemplo1.java`, and the right panel shows the console output.

```
1 package br.com.unitri.posjava.maps;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class Exemplo1 {
7
8     public static void main(String[] args) {
9
10         Map<String, String> mapa = new HashMap<Str
11
12         mapa.put("chave1", "valor1");
13         mapa.put("chave2", "valor2");
14         mapa.put("chave3", "valor3");
15         mapa.put("chave4", "valor4");
16
17         System.out.println(mapa);
18
19         System.out.println(mapa.get("chave1"));
20
21     }
22
23 }
```

The console output shows the result of the program execution:

```
<terminated> Exemplo1 (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/jav
[{chave2=valor2, chave1=valor1, chave4=valor4, chave3=valor3}
valor1
```

Análise assintótica

List implementations:

	get	add	contains	next	remove(0)	iterator.remove
ArrayList	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
CopyOnWriteArrayList	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$

Set implementations:

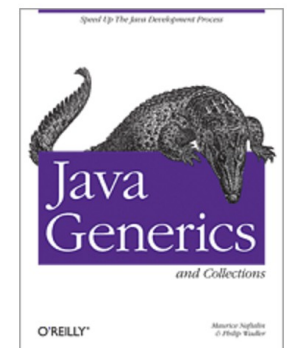
	add	contains	next	notes
HashSet	$O(1)$	$O(1)$	$O(h/n)$	h is the table capacity
LinkedHashSet	$O(1)$	$O(1)$	$O(1)$	
CopyOnWriteArraySet	$O(n)$	$O(n)$	$O(1)$	
EnumSet	$O(1)$	$O(1)$	$O(1)$	
TreeSet	$O(\log n)$	$O(\log n)$	$O(\log n)$	
ConcurrentSkipListSet	$O(\log n)$	$O(\log n)$	$O(1)$	

Map implementations:

	get	containsKey	next	Notes
HashMap	$O(1)$	$O(1)$	$O(h/n)$	h is the table capacity
LinkedHashMap	$O(1)$	$O(1)$	$O(1)$	
IdentityHashMap	$O(1)$	$O(1)$	$O(h/n)$	h is the table capacity
EnumMap	$O(1)$	$O(1)$	$O(1)$	
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$	
ConcurrentHashMap	$O(1)$	$O(1)$	$O(h/n)$	h is the table capacity
ConcurrentSkipListMap	$O(\log n)$	$O(\log n)$	$O(1)$	

Queue implementations:

	offer	peek	poll	size
PriorityQueue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
ConcurrentLinkedQueue	$O(1)$	$O(1)$	$O(1)$	$O(n)$
ArrayBlockingQueue	$O(1)$	$O(1)$	$O(1)$	$O(1)$
LinkedBlockingQueue	$O(1)$	$O(1)$	$O(1)$	$O(1)$
PriorityBlockingQueue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
DelayQueue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
LinkedList	$O(1)$	$O(1)$	$O(1)$	$O(1)$
ArrayDeque	$O(1)$	$O(1)$	$O(1)$	$O(1)$
LinkedBlockingDeque	$O(1)$	$O(1)$	$O(1)$	$O(1)$



Equals e hashCode

Condition	Required	Not Required (But Allowed)
<code>x.equals(y) == true</code>	<code>x.hashCode() == y.hashCode()</code>	
<code>x.hashCode() == y.hashCode()</code>		<code>x.equals(y) == true</code>
<code>x.equals(y) == false</code>		No <code>hashCode()</code> requirements
<code>x.hashCode() != y.hashCode()</code>	<code>x.equals(y) == false</code>	

Fonte: SCJP Sun Certified Programmer for Java 6 Study Guide

Equals e hashCode



The screenshot shows an IDE with two main panels. The left panel displays the source code of a Java application, and the right panel shows the console output.

Source Code (Exemplo2.java):

```
1 package br.com.unitri.posjava.sets;
2
3 import java.util.HashSet;
4 import java.util.Set;
5
6 public class Exemplo2 {
7
8     public static void main(String[] args) {
9
10         Set<Aluno> alunos = new HashSet<Aluno>();
11
12         Aluno a1 = new Aluno("Aluno 1", 11);
13         Aluno a2 = new Aluno("Aluno 2", 11);
14
15
16         alunos.add(a1);
17         alunos.add(a2);
18
19
20         System.out.println(alunos);
21     }
22 }
23
24
25
```

Console Output:

```
<terminated> Exemplo2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/bin/java (8
[Aluno [nome=Aluno 1, matricula=1]]
```

Equals e hashCode

```
38
39 @Override
40 public int hashCode() {
41     final int prime = 31;
42     int result = 1;
43     result = prime * result + ((matricula == null) ? 0 : matricula.hashCode());
44     return result;
45 }
46
47 @Override
48 public boolean equals(Object obj) {
49     if (this == obj)
50         return true;
51     if (obj == null)
52         return false;
53     if (getClass() != obj.getClass())
54         return false;
55     Aluno other = (Aluno) obj;
56     if (matricula == null) {
57         if (other.matricula != null)
58             return false;
59     } else if (!matricula.equals(other.matricula))
60         return false;
61     return true;
62 }
63
64
```


Equals e hashCode

The equals() Contract

Pulled straight from the Java docs, the `equals()` contract says

- It is **reflexive**. For any reference value `x`, `x.equals(x)` should return `true`.
- It is **symmetric**. For any reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is **transitive**. For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`.
- It is **consistent**. For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the object is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

Fonte: SCJP Sun Certified Programmer for Java 6 Study Guide

Equals e hashCode

The hashCode() Contract

Now coming to you straight from the fabulous Java API documentation for class `Object`, may we present (drum roll) the `hashCode()` contract:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in `equals()` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- It is NOT required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Fonte: SCJP Sun Certified Programmer for Java 6 Study Guide

Equals e hashCode

The hashCode() Contract

Now coming to you straight from the fabulous Java API documentation for class `Object`, may we present (drum roll) the `hashCode()` contract:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in `equals()` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- It is NOT required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Fonte: SCJP Sun Certified Programmer for Java 6 Study Guide

Generics

- Métodos genéricos e classes genéricas (e interfaces) permitem especificar, com uma única declaração de método, um conjunto de métodos relacionados ou, com uma única declaração de classe, um conjunto de tipos relacionados, respectivamente.
- Os genéricos também fornecem segurança de tipo em tempo de compilação que permite capturar tipos inválidos em tempo de compilação.
- Métodos sobrecarregados são frequentemente utilizados para realizar operações semelhantes em tipos diferentes de dados.

Generics

```
public void imprimir(Integer[] inteiros) {  
    for(Integer i : inteiros) {  
        System.out.println(i);  
    }  
}
```

```
public void imprimir(Long[] longs) {  
    for(Long i : longs) {  
        System.out.println(i);  
    }  
}
```

```
public void imprimir(String[] strings) {  
    for(String i : strings) {  
        System.out.println(i);  
    }  
}
```

- Códigos com semântica semelhante
- Repetição de código
- Difícil manutenção
- Etc...

Generics

```
public void imprimir(Integer[] inteiros) {  
    for(Integer i : inteiros) {  
        System.out.println(i);  
    }  
}  
  
public void imprimir(Long[] longs) {  
    for(Long i : longs) {  
        System.out.println(i);  
    }  
}  
  
public void imprimir(String[] strings) {  
    for(String i : strings) {  
        System.out.println(i);  
    }  
}
```



```
1 package br.com.unitri.posjava.generics;  
2  
3 public class ImpressoraGenerica<T> {  
4  
5     public void imprimir(T[] inteiros) {  
6  
7         for(T i : inteiros) {  
8             System.out.println(i);  
9         }  
10    }  
11 }  
12  
13 }  
14
```

Generics

- Se as operações realizadas por vários métodos sobrecarregados forem idênticas para cada tipo de argumento, os métodos sobrecarregados podem ser codificados mais compacta e convenientemente com um método genérico.
- Você pode escrever uma única declaração de método genérico que pode ser chamada com argumentos de tipos diferentes.
- Com base nos tipos dos argumentos passados para o método genérico, o compilador trata cada chamada de método apropriadamente.

Generics

Métodos genéricos

```
public static <X> void teste(X teste){  
    System.out.println(teste);  
}
```

- Além de uma classe genérica, você pode declarar um método genérico;
- O tipo genérico deve ser declarado entre < > e deve vir antes do retorno do método.

Generics

Wildcards

- `<?>` (mais conhecido como Unknown Wildcard, ou seja, Wildcard desconhecido)
- `<? extends A>`
- `<? super A>`

```
public void salva(List<?> dados) {  
    }  
  
public void salva2(List<? super Integer> dados){  
    }  
  
public void salva3(List<? extends Integer> dados){  
    }
```

```
public static <X> void salva(X teste){  
    }  
  
public static <X extends Integer> void salva2(X teste){  
    }
```