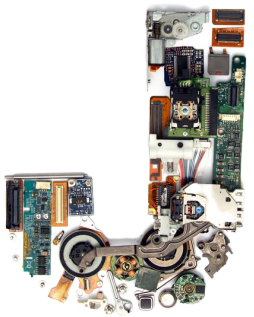
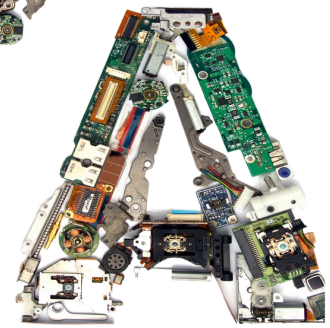
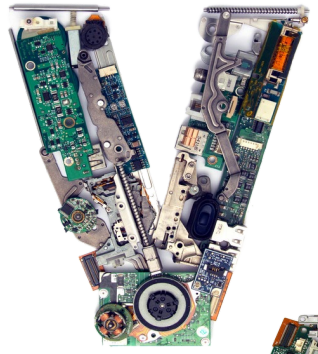
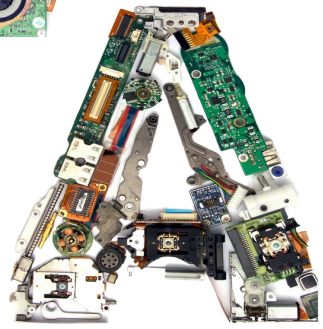




Programação Orientada a Objetos com Java e WEB



Genéricos e Coleções



Coleções

A API do Java fornece várias estruturas de dados predefinidas, chamadas *coleções* (*collection*), utilizadas para armazenar grupos de objetos relacionados.

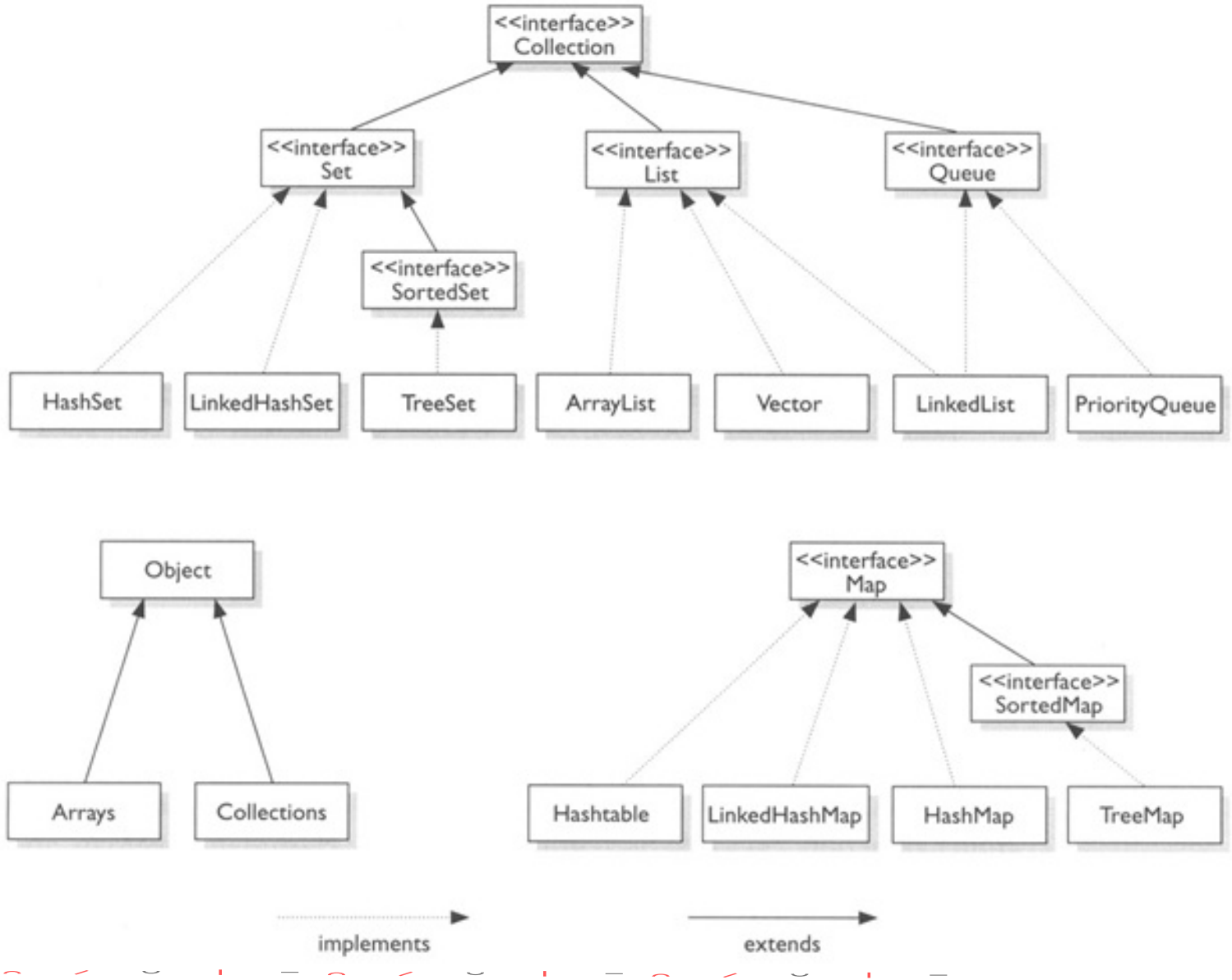
Coleção → objeto que agrupa vários elementos em uma unidade (objeto)

Essas classes fornecem métodos eficientes que organizam, armazenam e recuperam seus dados sem que seja necessário conhecer como os dados são armazenados.

Isso reduz o tempo de desenvolvimento de aplicativos.

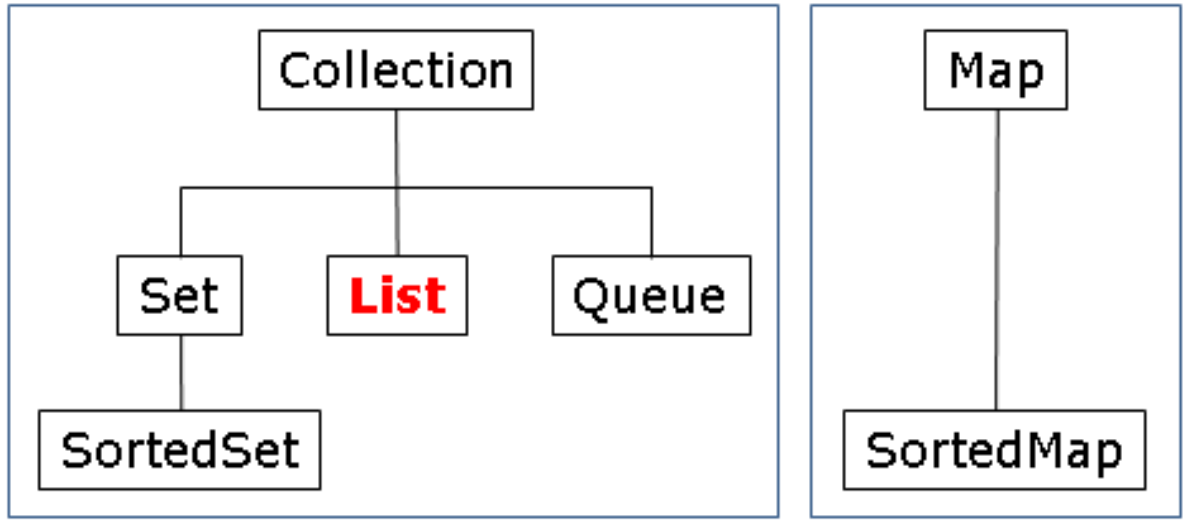
Os arrays utilizados até agora não alteram automaticamente seu tamanho em tempo de execução para acomodar elementos adicionais.

Framework Collection



os SELMINI – selmini@fiap.com.br

Framework Collection (resumindo)

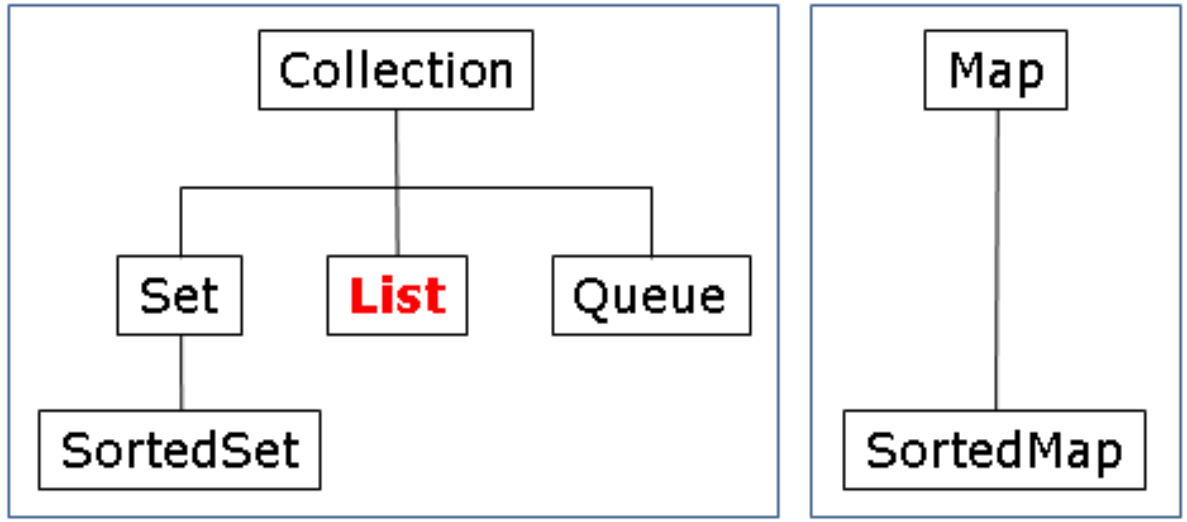


1. **Conjunto (*Set* e *SortedSet*):**
Uma coleção de elementos que modela a abstração matemática para conjuntos. Não mantém indexação e nem contagem dos elementos pertencentes. Cada elemento pertence ou não pertence ao conjunto (**não há elementos repetidos**). Podem ser mantidos ordenados (***SortedSet***) ou não.

2. **Lista (*List*):**
Uma coleção indexada de objetos às vezes chamada de **sequência**. Como nos vetores, índices de ***List*** são baseados em zero, isto é, o índice do primeiro elemento é zero. Além dos métodos herdados de ***Collection***, ***List*** fornece métodos para manipular elementos baseado na sua posição (ou índice) numérica na lista, remover determinado elemento, procurar as ocorrências de um dado elemento e percorrer sequencialmente (***ListIterator***) todos os elementos da lista. A interface ***List*** é implementada por várias classe, incluídas as classes ***ArrayList*** (implementada como vetor), ***LinkedList*** e ***Vector***.

Prof. Dr. Antonio Marcos SELMINI – selmini@fiap.com.br

Framework Collection (resumindo)



3. Fila (*Queue*):
Uma coleção utilizada para manter uma "fila" de elementos. Existe uma ordem linear para as filas que é a "ordem de chegada". As filas devem ser utilizadas quando os itens deverão ser processados de acordo com a ordem "PRIMEIRO-QUE-CHEGA, PRIMEIRO-ATENDIDO". Por esta razão as filas são chamadas de Listas FIFO, termo formado a partir de "*First-In, First-Out*".

4. Mapa (*Map* e *SortedMap*):
Um mapa armazena pares, chave e valor, chamados de itens. As chaves não podem ser duplicadas e são utilizadas para localizar um dado elementos associado. As chaves podem ser mantidas ordenadas (***SortedMap***) ou não.

Fonte: http://www.devmedia.com.br/explorando-a-classe-arraylist-no-java/24298

Interface List

É uma coleção “ordenada”. Semelhante a uma implementação de um array. Algumas vezes chamada sequência. Pode conter elementos duplicados.

Implementações: *ArrayList*, *LinkedList.*

ArrayList: ideal para pesquisa randômica.

LinkedList: ideal para pesquisa sequencial.

Classe ArrayList

A classe de coleção ArrayList<T> (**pacote java.util**) fornece uma solução conveniente ao problema dos arrays estáticos, pois ela pode alterar dinamicamente seu tamanho para acomodar mais elementos.

O T é um espaço reservado, que deverá ser substituído pelo tipo do elemento que será armazenado no ArrayList. Exemplos de declaração:

```
List<String> lista = new ArrayList<>();
Collections<String> lista = new ArrayList();
ArrayList<String> lista = new ArrayList<>();
ArrayList<Aluno> lista = new ArrayList<>();
```

Classe ArrayList

Método	Descrição
add(objeto)	Adiciona um elemento ao fim de ArrayList.
clear()	Remove todos os elementos de ArrayList.
contains(objeto)	Retorna true se ArrayList tiver o elemento especificado; do contrário, retorna false.
get(índice)	Retorna o elemento no índice(inteiro não negativo) especificado.
indexOf(objeto)	Retorna o índice da primeira ocorrência do elemento especificado em ArrayList. O índice inicia em zero!!
remove(índice)	Remove a primeira ocorrência do valor especificado.
remove(objeto)	Remove o elemento no índice especificado.
size()	Retorna o número de elementos armazenados no ArrayList.
trimToSize()	Reduz a capacidade de ArrayList de acordo com o número de elementos atual.

Classe ArrayList – exemplo

```java
public class Aluno {
    private int rm;
    private String nome;

    public Aluno(int rm, String nome) {
        this.rm = rm;
        this.nome = nome;
    }

    public String toString() {
        return rm+"\n"+nome;
    }
}
```

```java
import java.util.ArrayList;

public class TesteAluno {
    public static void main(String[] args) {

        List<Aluno> lista = new ArrayList<>();
        lista.add(new Aluno(5252, "Maria"));
        lista.add(new Aluno(2323, "Pedro"));
        Aluno a;
        for(int k = 0; k < lista.size(); k++) {
            a = lista.get(k);
            System.out.println(a);
        }
    }
}
```

Imprimindo listas

1. Usando o for tradicional

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

2. Usando o for genérico (for each)

```
for(Aluno aluno : lista) {
    System.out.println(aluno);
}
```

3. Usando o iterator

```
Iterator<Aluno> it = lista.iterator();
while(it.hasNext()) {
    System.out.println(it.next());
}
```

4. Usando lambda (recurso Java 8)

```
lista.forEach(nome -> {
    System.out.println(nome);
});
```

Ordenando listas

Como colocar a lista abaixo em ordem alfabética?

```
List<String> lista = new ArrayList<>();
lista.add("Marcos");
lista.add("Beatriz");
lista.add("Zileide");
lista.add("Ana");
System.out.println(lista); //método toString() da classe ArrayList é chamado
```

Saída no vídeo:

```
[Marcos, Beatriz, Zileide, Ana]
```

Ordenando listas

A classe Collections do pacote java.util apresenta uma série de métodos estáticos que operam ou retornam coleções.

Método estático sort():

métodos que são invocados utilizando o nome da classe (sem utilizar objetos)

static <T extends Comparable<? super T>>
void

sort(List<T> list) Sorts the specified list into ascending order, according to the natural ordering of its elements.

```
List<String> lista = new ArrayList<>();
lista.add("Marcos");
lista.add("Beatriz");
lista.add("Zileide");
lista.add("Ana");
Collections.sort(lista);
System.out.println(lista);
```

Saída no vídeo:

```
[Ana, Beatriz, Marcos, Zileide]
```

Ordenando listas

O método sort() é sobrecarregado e aceita um segundo parâmetro que é o critério de ordenação. O critério de ordenação é obtido a partir de um método estático da interface Comparator (pacote java.util).

static <T> void	sort(List<T> list, Comparator<? super T> c)Sorts the specified list according to the order induced by the specified comparator.

```
List<String> lista = new ArrayList<String>();
lista.add("Marcos");
lista.add("Beatriz");
lista.add("Zileide");
lista.add("Ana");
Collections.sort(lista, Comparator.reverseOrder());
System.out.println(lista);
```

Saída no vídeo:

```
[Zileide, Marcos, Beatriz, Ana]
```

Ordenando listas

Considere a classe **Aluno** definida abaixo. Como podemos ordenar uma lista de alunos em ordem alfabética?

```
public class Aluno {
    private int rm;
    private String nome;

    public Aluno(int rm, String nome) {
        this.rm = rm;
        this.nome = nome;
    }
}
```

Lista para ordenação:

```
List<Aluno> lista = new ArrayList<>();
lista.add(new Aluno(56, "Gustavo"));
lista.add(new Aluno(32, "Pedro"));
lista.add(new Aluno(15, "Gabriela"));
lista.add(new Aluno(89, "Beatriz"));
Collections.sort(lista);
System.out.println(lista);
```

Saída no vídeo:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method sort(List<T>) in the type Collections is not applicable for the arguments
(List<Aluno>)
```

Ordenando listas

Para que ocorra a ordenação da lista é necessário que a classe Aluno implemente o método compareTo() da interface Comparable<T>.

```
public class Aluno implements Comparable<Aluno> {
    private int rm;
    private String nome;

    @Override
    public int compareTo(Aluno aluno) {
        return this.nome.compareTo(aluno.getNome());
    }
}
```

Lista para ordenação:

```
List<Aluno> lista = new ArrayList<>();
lista.add(new Aluno(56, "Gustavo"));
lista.add(new Aluno(32, "Pedro"));
lista.add(new Aluno(15, "Gabriela"));
lista.add(new Aluno(89, "Beatriz"));
Collections.sort(lista);
System.out.println(lista);
```

Saída no vídeo:

```
[Aluno@7852e922, Aluno@4e25154f, Aluno@70dea4e, Aluno@5c647e05]
```

método toString() não foi sobrescrito

Ordenando listas

Sobrescrevendo o método toString() da classe Aluno.

```
public class Aluno implements Comparable<Aluno> {
    private int rm;
    private String nome;

    @Override
    public String toString() {
        return rm+" - "+nome;
    }

    @Override
    public int compareTo(Aluno aluno) {
        return this.nome.compareTo(aluno.getNome());
    }
}
```

Lista para ordenação:

```
List<Aluno> lista = new ArrayList<>();
lista.add(new Aluno(56, "Gustavo"));
lista.add(new Aluno(32, "Pedro"));
lista.add(new Aluno(15, "Gabriela"));
lista.add(new Aluno(89, "Beatriz"));
Collections.sort(lista);
System.out.println(lista);
```

Saída no vídeo: [89 - Beatriz, 15 - Gabriela, 56 - Gustavo, 32 - Pedro]

Ordenando listas

Como ordenar a lista de alunos pelo **atributo rm**? Alteração no método **compareTo()**:

```
@Override
public int compareTo(Aluno aluno) {
    if(this.rm > aluno.getRm()) {
        return 1;
    }
    else if(this.rm < aluno.getRm()) {
        return -1;
    }
    return 0;
}
```

Lista para ordenação:

```
List<Aluno> lista = new ArrayList<>();
lista.add(new Aluno(56, "Gustavo"));
lista.add(new Aluno(32, "Pedro"));
lista.add(new Aluno(15, "Gabriela"));
lista.add(new Aluno(89, "Beatriz"));
Collections.sort(lista);
System.out.println(lista);
```

Saída no vídeo: `[15 - Gabriela, 32 - Pedro, 56 - Gustavo, 89 - Beatriz]`

Ordenando listas

Método **compareTo()**:

O método **compareTo()** é parte da interface **Comparable** no Java e é usado para definir uma ordem natural para objetos de uma classe.

Esta interface é especialmente útil para ordenação de coleções, como listas, arrays, ou qualquer estrutura de dados que precise de ordenação.

A interface **Comparable** está no pacote **java.lang.**

Ordenando listas

Implementação do método **compareTo()**:

Quando uma classe implementa a interface **Comparable**, ela precisa fornecer uma implementação do método **compareTo()**. Este método compara o objeto atual com o objeto especificado para determinar a ordem.

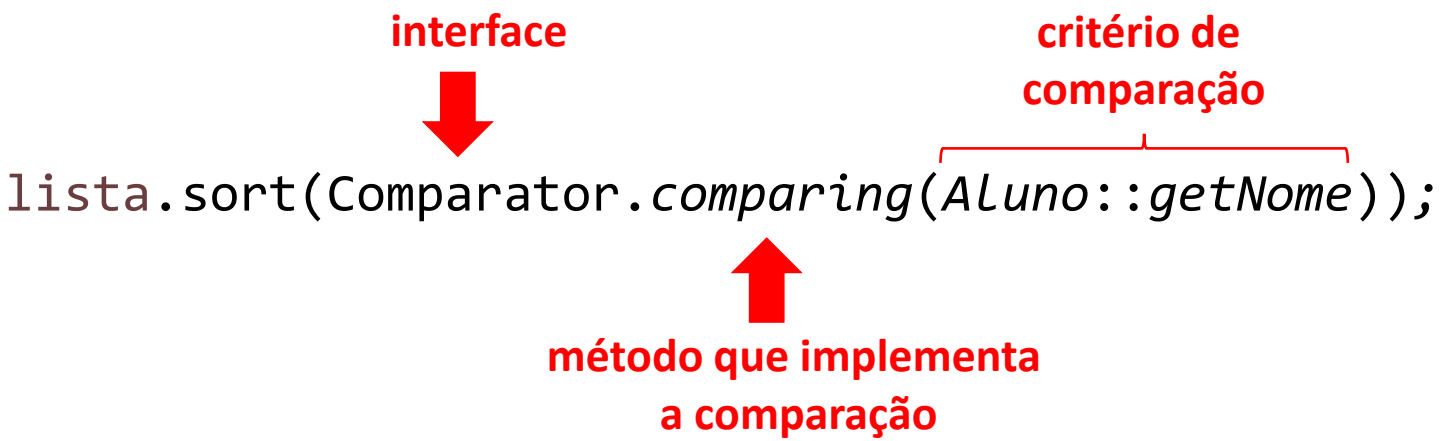
Valores de retorno do método compareTo():

O método **compareTo()** retorna um inteiro que indica a ordem relativa dos objetos comparados:

- **valor negativo** se o objeto atual é menor que o objeto especificado.
- **zero** se o objeto atual é igual ao objeto especificado.
- **valor positivo** se o objeto atual é maior que o objeto especificado.

Ordenando listas

Quando uma classe tem vários atributos, é possível estabelecer um critério de ordenação? Uma das formas é utilizar um recurso da versão 8 da linguagem. Exemplo:



Sintaxe introduzida no Java 8 → A::B → método B da classe A. O símbolo :: é chamado de referência de método.

Comparando dois objetos

Como verificar se dois objetos do tipo Aluno são iguais? O operador de igualdade (==) verifica as referências e não o conteúdo.

```
Aluno aluno1 = new Aluno(12, "Ana Beatriz");
Aluno aluno2 = new Aluno(12, "Ana Beatriz");
if(aluno1 == aluno2) {
    System.out.println("É mesmo aluno");
}
else {
    System.out.println("São alunos diferentes");
}
```

Saída no vídeo: São alunos diferentes

Comparando dois objetos

Quando dois objetos do tipo Aluno são iguais? Quando eles tiverem os mesmo atributos.
Uma alternativa seria criar um método para comparar os atributos dos dois objetos.

```
public static boolean comparar(Aluno aluno1, Aluno aluno2) {
    boolean iguais = false;
    if(aluno1.getRm() == aluno2.getRm() && aluno1.getNome().equals(aluno2.getNome())){
        iguais = true;
    }
    return iguais;
}
```

Comparando dois objetos

Outra alternativa seria sobrescrever o método **equals()** da classe **Object** na classe **Aluno**.

```
@Override
public boolean equals(Object o) {
    if((o instanceof Aluno) && ((Aluno)o).getRm() == this.rm) {
        return true;
    }
    return false;
}
```

```
Aluno aluno1 = new Aluno(12, "Ana Beatriz");
Aluno aluno2 = new Aluno(12, "Ana Beatriz");
if(aluno1.equals(aluno2)) {
    System.out.println("É mesmo aluno");
}
else {
    System.out.println("São alunos diferentes");
}
```

ArrayList ou LinkedList?

Existe diferença de desempenho entre as duas coleções?

```
public class Aula {
    private String conteudo;
    private int duracao;

    //demais métodos da classe
}
```

```
import java.util.List;

public class Disciplina {
    private String nome;
    private String professor;
    private List<Aula> listaAula;

    //demais métodos da classe
}
```

Para a lista de aulas da classe Disciplina podemos ter duas opções:

```
private List<Aula> listaAula = new ArrayList<>();
```

```
private List<Aula> listaAula = new LinkedList<>();
```

ArrayList ou LinkedList?

- Qual a diferença? **Desempenho**...
  - Um **ArrayList** usa internamente um **array** para fazer o armazenamento dos elementos. As operações de acesso são rápidas com o uso do método **get(indice)**, mas operações de inserção e remoção no início da estrutura são lentas.
  - A inserção no início de um **ArrayList** faz com que todos os elementos se desloquem para poder agrupar o novo elemento. A mesma observação vale para o processo de remoção.
  - Um **LinkedList** utiliza uma estrutura de dados chamada lista ligada que apresenta um bom desempenho para inserção no início, mas tem a desvantagem de ser lenta para buscar um elemento de forma aleatória, pois há a necessidade de percorrer todos os elementos.

Interface Set<>

A interface List<> é a mais utilizada na programação com sua implementação por meio da classe concreta ArrayList<>, mas existem outras interfaces que podem ser utilizadas.

A segunda interface mais utilizada é a interface Set<> implementada por meio da sua classe concreta HashSet<>.

```
Set<String> listaAlunos = new HashSet<>();
listaAlunos.add("Selmini");
listaAlunos.add("Ana");
listaAlunos.add("Beatriz");
listaAlunos.add("Maria");
listaAlunos.add("Carlos");
System.out.println(listaAlunos);
```

Saída no vídeo:

```
[Selmini, Ana, Maria, Carlos, Beatriz]
```

Não existe garantia da ordem em que os elementos foram inseridos.

Interface Set<>

- Quais as vantagens?
  - Não permite elementos duplicados.
  - A “velocidade” para encontrar um elemento no conjunto é mais rápida em relação ao ArrayList<>

```
Set<String> listaAlunos = new HashSet<>();
listaAlunos.add("Selmini");
listaAlunos.add("Ana");
listaAlunos.add("Beatriz");
listaAlunos.add("Selmini");
listaAlunos.add("Carlos");
System.out.println(listaAlunos);
```

Saída no vídeo:

```
[Selmini, Ana, Carlos, Beatriz]
```

Elementos repetidos são ignorados (não são inseridos)

Interface Set<>

Exemplo do emprego da interface Set<>

```
public class Disciplina {
    private String nome;
    private String professor;
    private List<Aula> listaAula = new ArrayList<>();
    private Set<Aluno> listaAluno = new HashSet<>();

    public void matricular(Aluno aluno) {
        listaAluno.add(aluno);
    }

    public boolean estaMatriculado(Aluno aluno) {
        return listaAluno.contains(aluno);
    }

    //demais métodos da classe
}
```

```
public class Aula {
    private String conteudo;
    private int duracao;

    //demais métodos da classe
}
```

```
public class Aluno {
    private int rm;
    private String nome;

    //demais métodos da classe
}
```

Interface Set<>

Como verificar se um aluno está matriculado em curso usando apenas o seu nome?

O método **contains()** da interface **Collection** utiliza o método **equals()** para fazer a comparação. Nesse caso a solução é sobrescrever o método **equals()** da classe **Aluno** para fazer a verificação apenas pelo atributo nome

```
@Override
public boolean equals(Object aluno) {
    Aluno aux = (Aluno) aluno;
    if(this.nome.equalsIgnoreCase(aux.getNome()) && aux.getRm() == this.rm) {
        return true;
    }
    return false;
}
```

se o valor do atributo nome for null uma exceção será lançada.

Interface Set<>

Como verificar se um aluno está matriculado em curso usando apenas o seu nome?

O método **contains()** da interface **Collection** utiliza o método **equals()** para fazer a comparação. Nesse caso a solução é sobrescrever o método **equals()** da classe **Aluno** para fazer a verificação apenas pelo atributo nome

```
@Override
public boolean equals(Object aluno) {
    Aluno aux = (Aluno) aluno;
    if(this.nome.equalsIgnoreCase(aux.getNome()) && aux.getRm() == this.rm) {
        return true;
    }
    return false;
}
```

se o valor do atributo nome for null uma exceção será lançada.

Interface Set<>

Para uma busca mais eficiente em um Set<> é necessário sobrescrever o método hashCode().

```
@Override
public int hashCode() {
    return this.nome.hashCode();
}
```

Map

Como localizar um aluno no curso pelo seu rm? Qual a frequência das buscas? Se a quantidade de alunos for grande, o processo terá um custo computacional alto.

Solução: utilizar a estrutura **Map**

```
public class Disciplina {
    private String nome;
    private String professor;
    private List<Aula> listaAula = new ArrayList<>();
    private Set<Aluno> listaAluno = new HashSet<>();
    private Map<Integer, Aluno> listaMatricula = new HashMap<>();
}
```

Map

Como inserir valores em um Map?

```
public class Disciplina {
    private String nome;
    private String professor;
    private List<Aula> listaAula = new ArrayList<>();
    private Set<Aluno> listaAluno = new HashSet<>();
    private Map<Integer, Aluno> listaMatricula = new HashMap<>();
}
```

```
public void matricular(Aluno aluno) {
    listaAluno.add(aluno);
    listaMatricula.put(aluno.getRm(), aluno);
}
```

Map

Como buscar um valor em um Map? Utiliza-se o método **get()**.

```
public class Disciplina {
    private String nome;
    private String professor;
    private List<Aula> listaAula = new ArrayList<>();
    private Set<Aluno> listaAluno = new HashSet<>();
    private Map<Integer, Aluno> listaMatricula = new HashMap<>();
}
```

```
public Aluno localizarAluno(int rm) {
    return listaMatricula.get(rm);
}
```

Map

É possível percorrer fazer a impressão dos elementos de um map a partir do valor da chave.

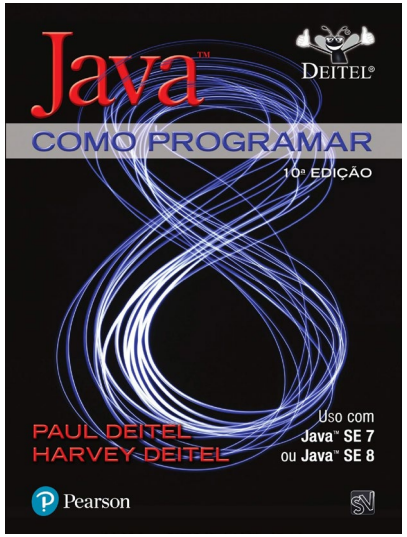
```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class Teste2 {
    public static void main(String[] args) {

        Map<Integer, Aluno> lista = new HashMap<Integer, Aluno>();

        lista.put(1, new Aluno(1, "Ana"));
        lista.put(2, new Aluno(2, "Pedro"));
        lista.put(3, new Aluno(3, "Bete"));
        lista.put(4, new Aluno(4, "João"));

        Set<Integer> listaRm = lista.keySet();
        for(Integer i : listaRm) {
            System.out.println(lista.get(i));
        }
    }
}
```

Retorna um conjunto (Set) contendo as chaves do Map

Referências

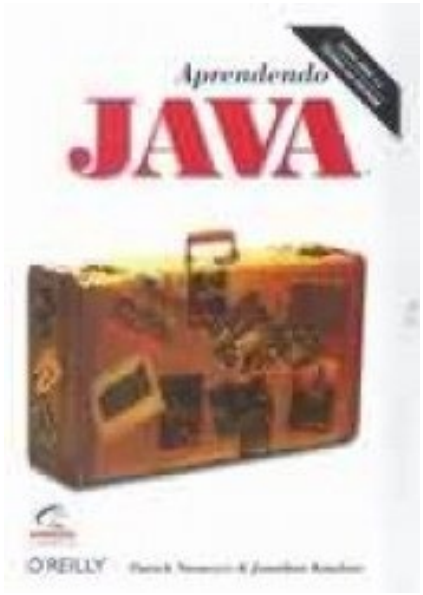
- DEITEL, H. M., DEITEL, P. J. **JAVA como programar**. 10ª edição. São Paulo: Prentice-Hall, 2010.

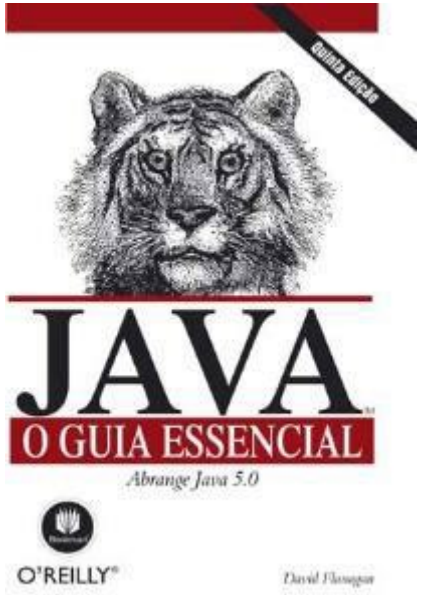
- SCHILDT, H. **Java para Iniciantes – Crie, Compile e Execute Programas Java Rapidamente**. 6ª Edição, Editora Bookman, Porto Alegre, RS, 2015.



Referências

- KNUDSEN, J., NIEMEYER, P. **Aprendendo Java**. Rio de Janeiro: Editora Elsevier Campus, 2000.
- FLANAGAN, D. **Java – o guia essencial**. Porto Alegre: Editora Bookman, 2006.





Referências

- ARNOLD, K., GOSLING, J., HOLMES, D., **Java programming language**. 4th Edition, Editora Addison-Wesley, 2005.
- JANDL JUNIOR, P. **Introdução ao Java**. São Paulo: Editora Berkeley, 2002.