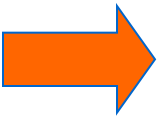


Programação Java

Prof. Vinicius Rosalen

O que é uma coleção ?

- Uma coleção (também denominada *container*)
 - É simplesmente um objeto que agrupa múltiplos elementos dentro de uma única unidade.
- Em outras palavras...
 - Conjunto de classes que permitem o agrupamento e processamento de grupos de objetos:
- São utilizadas para armazenar, recuperar e manipular dados,
 - Além de transmitir dados de um método para outro.



⇒ Hierarquia – Interfaces – Em poucas palavras...

⇒ List

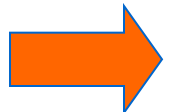
- É uma coleção onde a ordem é mantida, cada objeto pode ser manipulado através de seu índice

⇒ Set

- Uma coleção que não pode ter objetos duplicados

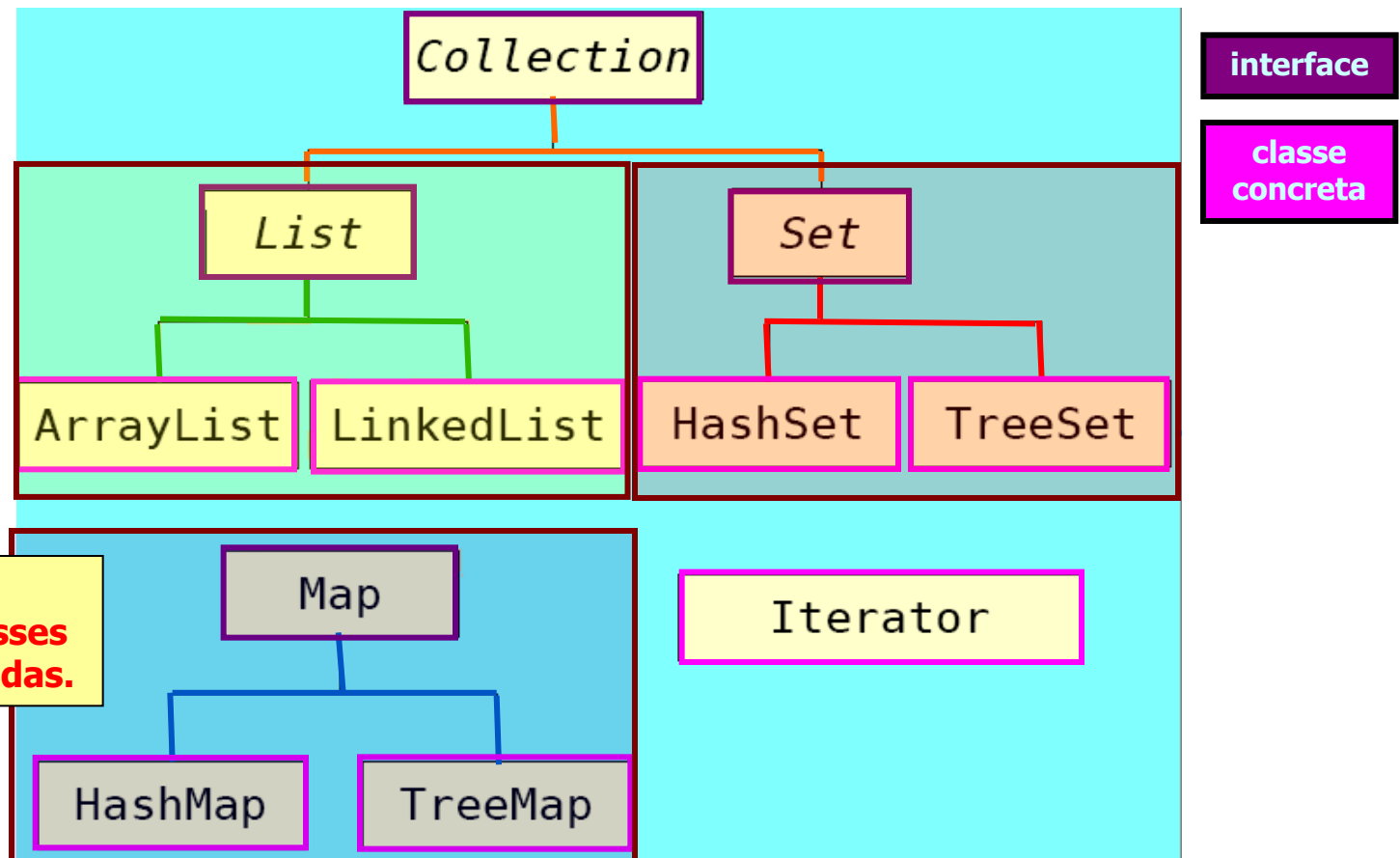
⇒ Map

- Mapeia objetos chaves para objetos, não são permitidas chaves duplicadas



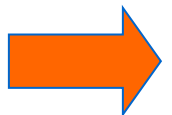
Hierarquia das Interfaces e suas Classes

Os elementos que compreendem a estrutura de coleções estão no pacote `java.util`.



Tipos Genéricos

- Blz,
 - Reforçado esses conceitos, podemos relembrar sobre uma **questão que é bastante utilizada** em desenvolvimento de código...
 - Toda a API de coleções foi adaptada para permitir o uso de **Tipos Genéricos**
 - Mas que raios é isso...



➡ Repare no uso de um parâmetro ao lado de List e ArrayList:

- Ele indica que nossa lista foi criada para trabalhar exclusivamente com objetos do tipo ContaCorrente.

➡ `List<ContaCorrente> contas = new ArrayList<ContaCorrente>();`

`contas.add(c1);`

`contas.add(c3);`

`contas.add(c2);`

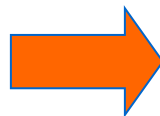
➡ `contas.get(i); // sem casting!`



Implica que o get não vai retornar um Object e sim uma ContaCorrente

➡ Isso também nos traz uma segurança em tempo de compilação:

➡ `contas.add("uma string"); // isso não compila mais!!`



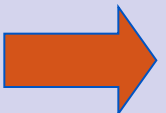
⇒ Genérico e Set

⇒ Antes

```
⇒ Set conjunto = new HashSet();  
   conjunto.add("item 1");  
   conjunto.add("item 2");  
   conjunto.add("item 3");  
  
   // retorna o iterator  
⇒ for(Object elemento : conjunto) {  
   ⇒ String palavra = (String) elemento;  
     System.out.println(palavra);  
}
```

⇒ Depois

```
⇒ Set<String> conjunto = new HashSet<String>();  
   conjunto.add("item 1");  
   conjunto.add("item 2");  
   conjunto.add("item 3");  
  
   // retorna o iterator  
⇒ for(String palavra : conjunto) {  
   ⇒ System.out.println(palavra);  
}
```



⇒ Genérico e Map

⇒ Assim como as coleções, um mapa é parametrizado.

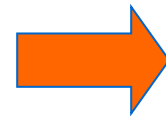
- O interessante é que ele recebe dois parâmetros:
 - A chave e o valor:

```
ContaCorrente c1 = new ContaCorrente();  
c1.deposita(10000);  
  
ContaCorrente c2 = new ContaCorrente();  
c2.deposita(3000);  
  
// cria o mapa  
⇒ Map<String, ContaCorrente> mapaDeContas = new  
HashMap<String, ContaCorrente>();  
  
// adiciona duas chaves e seus valores  
⇒ mapaDeContas.put("diretor", c1);  
mapaDeContas.put("gerente", c2);  
// qual a conta do diretor? (sem casting!)  
⇒ ContaCorrente contaDoDiretor = mapaDeContas.get("diretor");
```

⇒ Se você tentar colocar algo diferente de String na chave e ContaCorrente no valor... Vai ter um erro de compilação.

Implementação de relações – 1-N

- Blz,
 - Com essas estruturas podemos implementar, por exemplo, relações 1 pra vários...
 - Relembrando.....



Implementação de relações – 1-N e Coleções

- Para implementar relacionamentos desse tipo, baseado numa modelagem bidirecional, podemos:
 - ⇒ Inserir na classe que recebe a classe com multiplicidade N um atributo do tipo coleção (Set, por exemplo);
 - ⇒ Inserir os métodos get e set correspondentes ao atributo coleção;
 - ⇒ Inserir métodos para adicionar e remover objetos na coleção
 - ⇒ Inserir na classe que recebe a classe com multiplicidade 1 um atributo do tipo correspondente;

Implementação de relações – 1-N e Coleções

Socio
<ul style="list-style-type: none"> - nome : String - dependentes : Set
<ul style="list-style-type: none"> + Socio() + Socio(nome : String) + getNome() : String + setNome(nome : String) : void + getDependentes() : Set + setDependentes(dependentes : Set) : void + adicionarDependente(dependente : Dependente) : void + removerDependente(dependente : Dependente) : void
+ adicionarDependente(): void

Dependente
<ul style="list-style-type: none"> - nome : String - socio : Socio
<ul style="list-style-type: none"> + Dependente() + Dependente(nome : String) + Dependente(nome : String, socio : Socio) + getNome() : String + setNome(nome : String) : void + getSocio() : Socio + setSocio(socio : Socio) : void

1 0..*

```
1 import java.util.HashSet;
2 import java.util.Set;
```

```
4 public class Socio {
5     private String nome;
```

//Inserir na classe que recebe a classe com multiplicidade N um atributo
// do tipo colecao (Set, por exemplo);
private Set<Dependente> dependentes;

```
10 public Socio(){
11     super();
12     this.dependentes = new HashSet<Dependente>();
```

```
13 }
14 public Socio(String nome)
19 public void setNome(String nome)
22 public String getNome()
```

// Inserir os métodos get e set correspondentes ao atributo coleção;
private void setDependentes(Set<Dependente> dependentes)
public Set<Dependente> getDependentes()

```
45 public void adicionarDependente(Dependente dependente){
46     this.getDependentes().add(dependente);
47 }
48 public void removerDependente(Dependente dependente){
49     this.getDependentes().remove(dependente);
50 }
51 }
```

Implementação de relações – 1-N e Coleções

Socio
<ul style="list-style-type: none"> nome : String dependentes : Set
<ul style="list-style-type: none"> + Socio() + Socio(nome : String) + getNome() : String + setNome(nome : String) : void + getDependentes() : Set + setDependentes(dependentes : Set) : void + adicionarDependente(dependente : Dependente) : void + removerDependente(dependente : Dependente) : void

Dependente
<ul style="list-style-type: none"> nome : String socio : Socio
<ul style="list-style-type: none"> + Dependente() + Dependente(nome : String) + Dependente(nome : String, socio : Socio) + getNome() : String + setNome(nome : String) : void + getSocio() : Socio + setSocio(socio : Socio) : void

**// Inserir na classe que recebe
a classe com multiplicidade 1 um
atributo do tipo correspondente;**

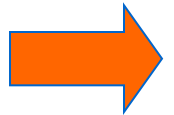
```

1  public class Dependente {
2      private String nome;
3      private Socio socio;
4
5      public Dependente(){}
6      public Dependente(String nome){}
7
8      public Dependente(String nome, Socio socio){
9          super();
10         this.nome = nome;
11         this.setSocio(socio);
12     }
13
14     public void setNome(String nome){}
15     public String getNome(){}
16
17     public Socio getSocio(){}
18
19     // Aqui estamos vinculando ao dependente qual é o seu socio, ou
20     // seja, qual socio ele está relacionado...
21     // Para isso vamos adicionar esse dependente na lista de
22     // dependentes validos do socio em questão...
23     public void setSocio(Socio socio){
24         // Linha 36: opcional, pois o Set já trata isso
25         // if(this.socio!=null){
26         //     this.socio.removerDependente(this);
27         // }
28         this.socio = socio;
29         if(this.socio!=null){
30             this.socio.adicionarDependente(this);
31         }
32     }
33
34     @Override
35     public boolean equals(Object obj){}
36
37     @Override
38     public int hashCode(){}
39 }

```

Ordenação de coleções

- Blz,
 - Uma vez entendido todas essas coisas...
 - Vamos conversar um pouco sobre o nosso foco de hoje que é sobre a ordenação dessas coleções de objetos....



Ordenação de coleções

→ Java já implementa alguns algoritmo de ordenação:

- Coleções ordenadas: TreeSet, TreeMap;
- Collections.sort() para coleções;
- Arrays.sort() para vetores.

→ Por exemplo

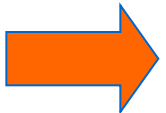
```
1  import java.util.*;
2
3  public class Ordem1 {
4      public static void main(String args[]) {
5
6          → List lista = new ArrayList();
7              lista.add("Sergio");
8              lista.add("Vinicius");
9              lista.add("Paulo");
10             lista.add("Guilherme");
11
12             System.out.println(lista);
13             → Collections.sort(lista);
14             System.out.println(lista);
15
16         }
17     }
```

```
[Sergio, Vinicius, Paulo, Guilherme]
[Guilherme, Paulo, Sergio, Vinicius]
```

→ Observe que primeiro a lista é impressa na ordem de inserção e, depois de chamar o sort, ela é impressa em ordem alfabética

Ordenação de coleções

- ➡ Mas toda coleção em Java pode ser de qualquer tipo de objeto
 - Por exemplo: ContaCorrente.
 - ➡ E se quisermos ordenar uma lista de ContaCorrente?
 - ➡ Em que ordem a classe Collections ordenará?
 - ➡ Pelo saldo? Pelo nome do correntista?
- ➡ Sempre que falamos em ordenação, precisamos pensar em um **critério de ordenação**,
 - Uma forma de determinar qual elemento vem antes de qual.
- ➡ É necessário instruir o Java sobre como **comparar** nossas Objetos
 - A fim de determinar uma ordem na coleção.



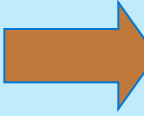
Ordenação de coleções - Comparable

➡ Para que esse tipo de ordenação funcione,
– É preciso que os objetos implementem a interface Comparable

➡ Esta interface define a operação de comparação do próprio objeto com outro,
– Usado para definir a ordem natural dos elementos de uma coleção.

➡ Define o método `compareTo(Object obj)`:
➡ Compara o objeto atual (this) com o objeto informado (obj);

- Retorna 0 se `this = obj`;
- Retorna um número negativo se `this < obj`;
- Retorna um número positivo se `this > obj`.

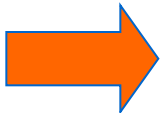


Ordenação de coleções - Comparable

⇒ O que o código está fazendo?

⇒ E se quisermos ordenar pela idade?

```
1  import java.util.*;
2
3  class Pessoa implements Comparable {
4      private String nome;
5      private int idade;
6
7      public Pessoa(String n, int i) {
8          nome = n; idade = i;
9      }
10
11     public String toString() {
12         return nome + ", " + idade + " ano(s)";
13     }
14
15     public int compareTo(Object o) {
16         return nome.compareTo(((Pessoa)o).nome);
17     }
18 }
19
20 public class Ordem2 {
21     public static void main(String[] args) {
22
23         List pessoas = new ArrayList();
24         pessoas.add(new Pessoa("Fulano", 20));
25         pessoas.add(new Pessoa("Beltrano", 18));
26         pessoas.add(new Pessoa("Cicrano", 23));
27
28         Collections.sort(pessoas);
29
30         for (Object o : pessoas)
31             System.out.println(o);
32     }
33 }
34
```



Ordenação de coleções - Comparable

⇒ O que o código está fazendo?

⇒ E se quisermos ordenar pela idade em ordem decrescente?

```
1  import java.util.*;
2
3  class Pessoa implements Comparable {
4      private String nome;
5      private int idade;
6
7      public Pessoa(String n, int i) {
8          nome = n; idade = i;
9      }
10
11     public String toString() {
12         return nome + ", " + idade + " ano(s)";
13     }
14
15     public int compareTo(Object o) {
16         // return nome.compareTo(((Pessoa)o).nome);
17         if(this.idade < ((Pessoa)o).idade) {
18             return -1;
19         }
20         if(this.idade > ((Pessoa)o).idade) {
21             return 1;
22         }
23         return 0;
24     }
25 }
26
27 public class Ordem2 {
28     public static void main(String[] args) {
29
30         List pessoas = new ArrayList();
31         pessoas.add(new Pessoa("Fulano", 20));
32         pessoas.add(new Pessoa("Beltrano", 18));
33         pessoas.add(new Pessoa("Cicrano", 23));
34
35         Collections.sort(pessoas);
36
37         for (Object o : pessoas)
38             System.out.println(o);
39     }
40 }
41
```

Ordenação de coleções - Comparable

⇒ Resumindo

⇒ Com o código anterior, nossa classe tornou-se “comparável”

⇒ Dados dois objetos da classe, conseguimos dizer se um objeto é maior, menor ou igual ao outro, segundo algum critério por nós definido.

- No nosso caso, a comparação foi feita baseando no nome e depois na idade.

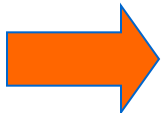


Ordenação de coleções - Comparable

⇒ Resumindo

- ⇒ **IMPORTANTE:** Repare que o critério de ordenação é **totalmente aberto, definido pelo programador.**
- Se quisermos ordenar por outro atributo (ou até por uma combinação de atributos), basta modificar a implementação do método `compareTo` na classe.

- ⇒ Agora sim, quando chamarmos o método `sort` de `Collections` ele saberá como fazer a ordenação da lista;
- Ele usará o critério que **definimos no método `compareTo`**



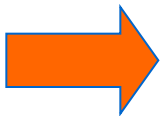
Ordenação de coleções - Comparator

➡ E o que acontece quando não podemos criar comparadores pois não posso mexer no código??

➡ Podemos utilizar a **interface Comparable** quando os objetos a serem adicionados não podem ser modificados....

– biblioteca de terceiros, por exemplo

➡ Importante: Esta interface define a operação de comparação entre dois objetos por um objeto externo.



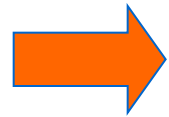
Ordenação de coleções - Comparator

➡ Para trabalhar com esse interface....

- É necessário implementar `java.util.Comparator`;

➡ Esta interface define o método `compare(Object a, Object b)` e retorna:

- Número negativo, se o primeiro $a < b$;
- Zero, se $a = b$;
- Número positivo se $a > b$.



Ordenação de coleções - Comparator

➡ O que o código está fazendo?

➡ E se quisermos ordenar pelo nome?

```
1  import java.util.*;
2
3  class Pessoa {
4      public String nome;
5      public int idade;
6
7      public Pessoa() {}
8
9      public Pessoa(String n, int i) {
10         nome = n; idade = i;
11     }
12
13     public String toString() {
14         return nome + ", " + idade + " ano(s)";
15     }
16 }
17
18 class PessoaComparator implements Comparator {
19
20     //Número negativo, se o1 < o2;
21     //Zero, se o1 = o2;
22     //Número positivo se o1 > o2.
23     public int compare(Object o1, Object o2) {
24         return (((Pessoa)o1).idade - ((Pessoa)o2).idade);
25     }
26 }
27
28 public class Ordem3 {
29     public static void main(String[] args) {
30
31         List pessoas = new ArrayList();
32         pessoas.add(new Pessoa("Fulano", 20));
33         pessoas.add(new Pessoa("Beltrano", 18));
34         pessoas.add(new Pessoa("Cicrano", 23));
35
36         Collections.sort(pessoas, new PessoaComparator());
37
38         for (Object o : pessoas)
39             System.out.println(o);
40     }
41 }
```

Ordenação de coleções - Comparator

➡ O que o código
está fazendo?

```
1  import java.util.*;
2
3  class Pessoa {
4      public String nome;
5      public int idade;
6
7      public Pessoa() {}
8
9      public Pessoa(String n, int i) {
10         nome = n; idade = i;
11     }
12
13     public String toString() {
14         return nome + ", " + idade + " ano(s)";
15     }
16 }
17
18 class PessoaComparator implements Comparator {
19     //Número negativo, se o1 < o2;
20     //Zero, se o1 = o2;
21     //Número positivo se o1 > o2.
22     public int compare(Object o1, Object o2) {
23         // return (((Pessoa)o1).idade - ((Pessoa)o2).idade);
24         ➡ return ((Pessoa)o1).nome.compareTo(((Pessoa)o2).nome);
25     }
26 }
27
28
29 public class Ordem3 {
30     public static void main(String[] args) {
31
32         List pessoas = new ArrayList();
33         pessoas.add(new Pessoa("Fulano", 20));
34         pessoas.add(new Pessoa("Beltrano", 18));
35         pessoas.add(new Pessoa("Cicrano", 23));
36
37         ➡ Collections.sort(pessoas, new PessoaComparator());
38
39         for (Object o : pessoas)
40             System.out.println(o);
41     }
42 }
43
```


Tipos Genéricos e ordenação

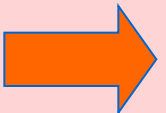
- ➡ Pra fechar... Outra coisa legal que podemos fazer é:
- ➡ Definir a ordenação dos objetos para o tipo de objetos que estamos trabalhando.

➡ Antes

```
➡ class Pessoa implements Comparable {  
    private String nome;  
    ➡ public int compareTo(Object o) {  
        ➡ Pessoa p = (Pessoa)o;  
        return nome.compareTo(p.nome);  
    }  
}
```

➡ Depois

```
// Com generics:  
➡ class Pessoa implements Comparable<Pessoa> {  
    private String nome;  
    ➡ public int compareTo(Pessoa o) {  
        return nome.compareTo(o.nome);  
    }  
}
```



⇒ Outro exemplo de Genérico usando Comparable

⇒ Observe que o método compareTo recebe um objeto ContaCorrente e não um Object

```
⇒ public class ContaCorrente extends Conta implements Comparable<ContaCorrente>  
{
```

```
    ⇒ public int compareTo(ContaCorrente outra) {
```

```
        if(this.saldo < outra.saldo) {  
            return -1;  
        }
```

```
        if(this.saldo > outra.saldo) {  
            return 1;  
        }
```

```
        return 0;
```

```
    }
```

```
}
```



Tipos Genéricos e ordenação

Qual a saída do código abaixo?

```

6  class Presidente {
7      String nome;
8      int inicio, fim;
9      public Presidente(String n, int i, int f) {
10         nome = n; inicio = i; fim = f;
11     }
12     public String toString() {
13         return nome + ": de " + inicio + " ate " + fim + "\n";
14     }
15 }
16
17 class ComparatorPresidente implements Comparator<Presidente> {
18
19     // Antes
20     public int compare(Object o1, Object o2) {
21         Presidente p1 = (Presidente) o1;
22         Presidente p2 = (Presidente) o2;
23
24         public int compare(Presidente p1, Presidente p2) {
25             return p1.inicio - p2.inicio;
26         }
27     }
28
29 class OrdenaPresidentesGenerics {
30
31     public static void main(String[] args) {
32
33         List teste = new ArrayList();
34         teste.add (new Presidente ("Luis Inacio", 2006, 2010));
35         teste.add (new Presidente ("Luis Inacio", 2002, 2005));
36         teste.add (new Presidente ("Fernando Henrique", 1998, 2001));
37         teste.add (new Presidente ("Fernando Henrique", 1994, 1997));
38
39         Comparator crescente = new ComparatorPresidente();
40         Comparator decrescente = Collections.reverseOrder(crescente);
41
42         // Em ordem crescente do início do mandato
43         Collections.sort (teste, crescente);
44         System.out.println (teste);
45
46         // Em ordem decrescente do fim do mandato
47         Collections.sort (teste, decrescente);
48         System.out.println (teste);
49     }
50 }
51

```

O interessante é observar que podemos mudar o critério de ordenação simplesmente modificando o comparator

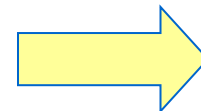


Exercício Junto....

- Blz, entendemos muita coisa hoje.....

➡ Vamos agora construir juntos um exemplo que tenha:

- Comparable
- Comparator
- Generics





Exercício Junto....

- Primeiro implemente o código ao lado.
- Logo em seguida modifique para o uso de Generics

```

1  import java.util.*;
2
3  class Pessoa implements Comparable {
4      private String nome;
5      private int idade;
6
7      public Pessoa(String n, int i) {
8          nome = n; idade = i;
9      }
10
11     public String toString() {
12         return nome + ", " + idade + " ano(s)";
13     }
14
15     public int compareTo(Object o) {
16         // return nome.compareTo(((Pessoa)o).nome);
17         if(this.idade < ((Pessoa)o).idade) {
18             return -1;
19         }
20         if(this.idade > ((Pessoa)o).idade) {
21             return 1;
22         }
23         return 0;
24     }
25 }
26
27 public class Ordem2 {
28     public static void main(String[] args) {
29
30         List pessoas = new ArrayList();
31         pessoas.add(new Pessoa("Fulano", 20));
32         pessoas.add(new Pessoa("Beltrano", 18));
33         pessoas.add(new Pessoa("Cicrano", 23));
34
35         Collections.sort(pessoas);
36
37         for (Object o : pessoas)
38             System.out.println(o);
39     }
40 }
41

```

Exercício Junto....

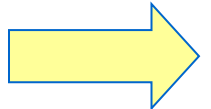
- Primeiro implemente o código ao lado.
- Logo em seguida modifique para o uso de Generics

```

1  import java.util.*;
2
3  class Pessoa {
4      public String nome;
5      public int idade;
6
7      public Pessoa() {}
8
9      public Pessoa(String n, int i) {
10         nome = n; idade = i;
11     }
12
13     public String toString() {
14         return nome + ", " + idade + " ano(s)";
15     }
16 }
17
18 class PessoaComparator implements Comparator {
19
20     //Número negativo, se o1 < o2;
21     //Zero, se o1 = o2;
22     //Número positivo se o1 > o2.
23     public int compare(Object o1, Object o2) {
24         // return (((Pessoa)o1).idade - ((Pessoa)o2).idade);
25         return ((Pessoa)o1).nome.compareTo(((Pessoa)o2).nome);
26     }
27 }
28
29 public class Ordem3 {
30     public static void main(String[] args) {
31
32         List pessoas = new ArrayList();
33         pessoas.add(new Pessoa("Fulano", 20));
34         pessoas.add(new Pessoa("Beltrano", 18));
35         pessoas.add(new Pessoa("Cicrano", 23));
36
37         Collections.sort(pessoas, new PessoaComparator());
38
39         for (Object o : pessoas)
40             System.out.println(o);
41     }
42 }
43

```

- Blz... Agora é hora de exercitar.....
 - Tente resolver os seguintes problemas...
 - Em dupla
 - Apresentar ao professor no final da aula
- Faça o JAVADOC de todos os exercícios!!!



Exercício

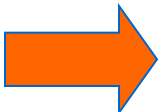
- Escreva um programa Java que:
 - Crie 5 objetos Circulo de tamanho diferentes
 - Calcular a sua área ($A = \pi \times r^2$)
 - Insira-os em uma lista
 - E depois percorra a lista imprimindo em ordem crescente por área os círculo armazenados.

Exercício

- Escreva um programa que leia nomes, idades e alturas de várias pessoas e armazene numa lista.
- Em seguida, imprima o conteúdo desta lista ordenado:
 - Por nome,
 - Depois ordenado por idade
 - E por fim ordenado por altura.

- No código a seguir, compile, execute no computador e analise os resultados.. **Estude na prática!!!**

```
1 // Sort a list using the custom Comparator class TimeComparator.
2 import java.util.List;
3 import java.util.ArrayList;
4 import java.util.Collections;
5
6 public class Sort3
7 {
8     public void printElements()
9     {
10         List< Time2 > list = new ArrayList< Time2 >(); // create List
11
12         list.add( new Time2( 6, 24, 34 ) );
13         list.add( new Time2( 18, 14, 58 ) );
14         list.add( new Time2( 6, 05, 34 ) );
15         list.add( new Time2( 12, 14, 58 ) );
16         list.add( new Time2( 6, 24, 22 ) );
17
18         // output List elements
19         System.out.printf( "Unsorted array elements:\n%s\n", list );
20
21         // sort in order using a comparator
22         Collections.sort( list, new TimeComparator() );
23
24         // output List elements
25         System.out.printf( "Sorted list elements:\n%s\n", list );
26     } // end method printElements
27
28     public static void main( String args[] )
29     {
30         Sort3 sort3 = new Sort3();
31         sort3.printElements();
32     } // end main
33 } // end class Sort3
34
```



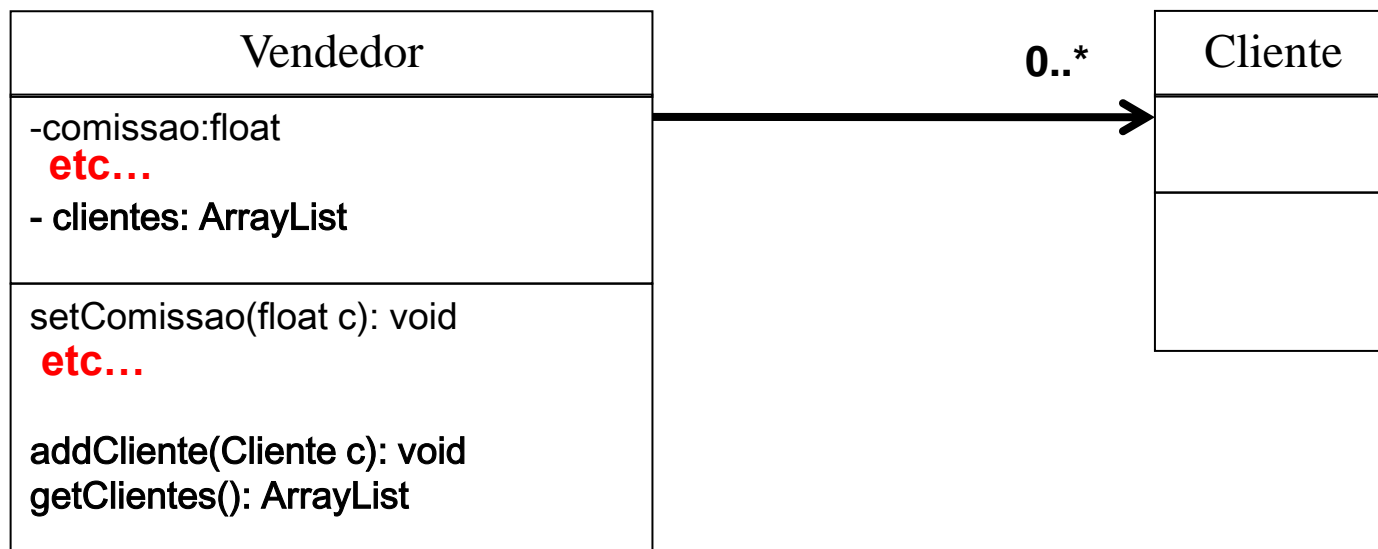
- No código a seguir, compile, execute no computador e analise os resultados.. Estude na prática!!!

```
1 // Custom Comparator class that compares two Time2 objects.
2 import java.util.Comparator;
3
4 public class TimeComparator implements Comparator< Time2 >
5 {
6     public int compare( Time2 time1, Time2 time2 )
7     {
8         int hourCompare = time1.getHour() - time2.getHour(); // compare hour
9
10        // test the hour first
11        if ( hourCompare != 0 )
12            return hourCompare;
13
14        int minuteCompare =
15            time1.getMinute() - time2.getMinute(); // compare minute
16
17        // then test the minute
18        if ( minuteCompare != 0 )
19            return minuteCompare;
20
21        int secondCompare =
22            time1.getSecond() - time2.getSecond(); // compare second
23
24        return secondCompare; // return result of comparing seconds
25    } // end method compare
26 } // end class TimeComparator
```

Exercício

- Relação de 1 para n, unidirecional

- Implemente a associação entre as classes Vendedor e Cliente. Suponha que um determinado vendedor possuí vários clientes.
- Escreva um programa de teste capaz de verificar a implementação da relação.
- Imprima uma listagem em ordem crescente de comissão.





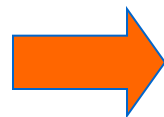
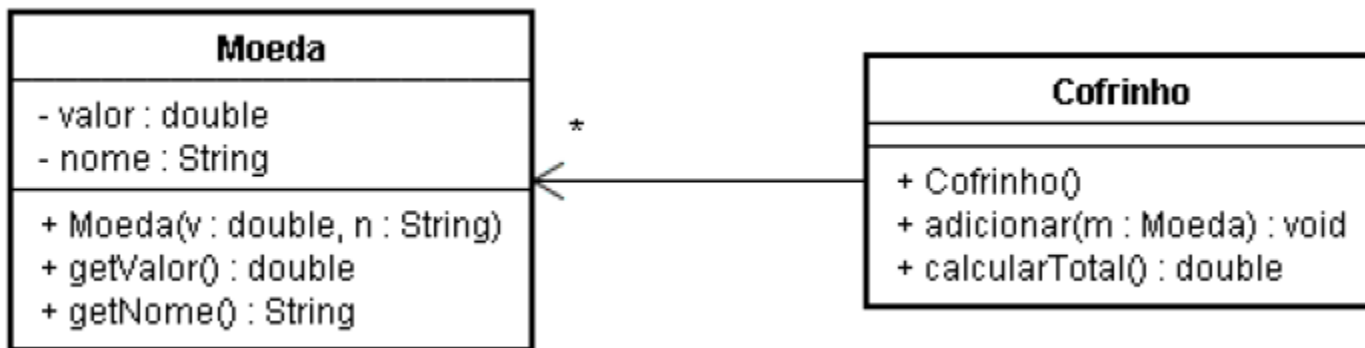
Exercício



Exercício

- Implementar um cofrinho de moedas com a capacidade de receber moedas e calcular o total depositado no cofrinho.
 - Implementa uma coleção de Moeda como uma lista.
 - Use o ArrayList
 - Faça um classe de teste

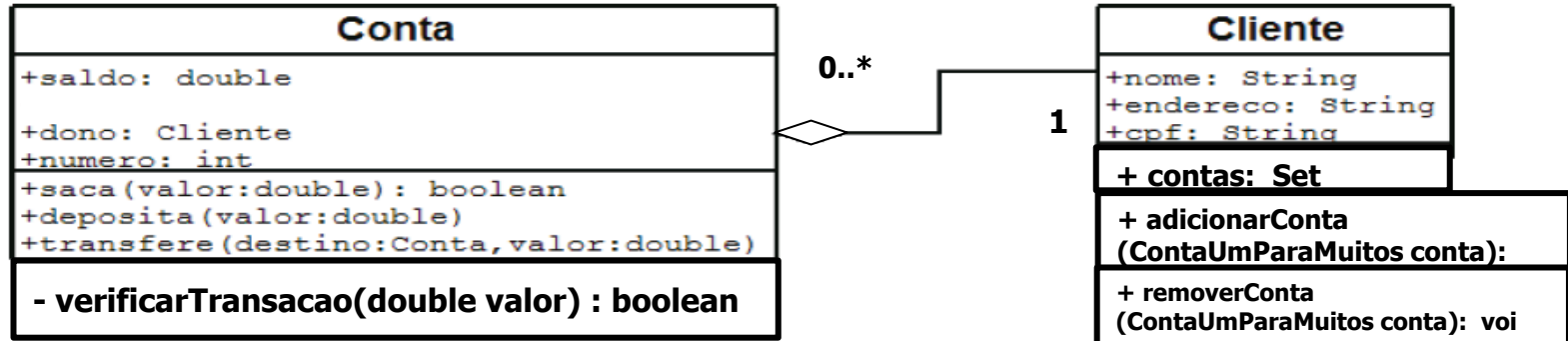
```
public class Cofrinho {  
    private List<Moeda> moedas;  
    public Cofrinho() {}  
    public .... get/setMoeda(...) {...}  
}
```



Exercício

- Altere a classe Cofrinho de modo que ela implemente métodos para:
 - Contar o número de moedas armazenadas
 - Contar o número de moedas de um determinado valor
 - Informar qual a moeda de maior valor
 - Imprima uma listagem por ordem decrescente de valor

- Implemente as classes abaixo de acordo com as orientações e imprima 2 listagem: 1 por ordem alfabética de nome dos clientes e 1 por ordem crescente de saldo nas contas

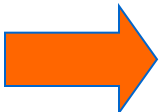


- Criar as classes Conta e Cliente com os atributos básicos.
- Gerar os gets e set para todos atributos e construtores padrão.
 - Criar o equals e hashCode na classe Conta
 - this.contas = new HashSet<Conta>() no construtor do Cliente
- Criar na classe Cliente (a que tem a coleção) os métodos para adicionar e remover objetos na coleção.
- Alter o método SetDono(..) na classe Conta para que a Conta possa ser vinculada a coleção de contas do Cliente
- Agora podemos criar classe Principal, criando um cliente, chamando o método para adicionar suas contas e depois 1: Listar quais são as contas do Cliente; 2: Listar qual é o Cliente associado as Contas

Exercício

- Faça uma classe ContaPoupanca implementar a interface Comparable<ContaPoupanca>.
 - Use uma lista
 - Utilize o critério de ordenar pelo número da conta ou pelo seu saldo
 - Dê uma olhada no material para exemplos..
 - Dica:

```
public class ContaPoupanca extends Conta implements  
Comparable<ContaPoupanca>
```



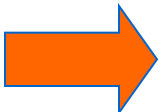
Exercício

- Crie uma classe TestaOrdenacao, instancie diversas contas e adicione-as numa List<ContaPoupanca>.
 - Use o Collections.sort() nessa lista
 - Faça um laço para imprimir todos os saldos das contas na lista já ordenada:

```
List<ContaPoupanca> contas = new ArrayList<ContaPoupanca>();
```

- Dica:

```
ContaPoupanca c1 = new ContaPoupanca();  
c1.deposita(150);  
contas.add(c1);  
  
ContaPoupanca c2 = new ContaPoupanca();  
c2.deposita(100);  
contas.add(c2);  
  
ContaPoupanca c3 = new ContaPoupanca();  
c3.deposita(200);  
  
contas.add(c3);  
  
Collections.sort(contas);
```





Exercício

- Mude o critério de comparação da sua ContaPoupanca.
 - Adicione um atributo nomeDoCliente na sua classe (caso ainda não exista algo semelhante),
 - Tente mudar o compareTo para que uma lista de ContaPoupanca seja ordenada alfabeticamente pelo atributo nomeDoCliente.

- No código a seguir, compile, execute no computador e analise os resultados.. Estude na prática!!!

```
1 // Generic method maximum returns the largest of three objects.
2
3 public class MaximumTest
4 {
5     // determines the largest of three Comparable objects
6     public static < T extends Comparable< T > > T maximum( T x, T y, T z )
7     {
8         T max = x; // assume x is initially the largest
9
10        if ( y.compareTo( max ) > 0 )
11            max = y; // y is the largest so far
12
13        if ( z.compareTo( max ) > 0 )
14            max = z; // z is the largest
15
16        return max; // returns the largest object
17    } // end method maximum
18
19    public static void main( String args[] )
20    {
21        System.out.printf( "Maximum of %d, %d and %d is %d\n\n", 3, 4, 5,
22            maximum( 3, 4, 5 ) );
23        System.out.printf( "Maximum of %.1f, %.1f and %.1f is %.1f\n\n",
24            6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );
25        System.out.printf( "Maximum of %s, %s and %s is %s\n", "pear",
26            "apple", "orange", maximum( "pear", "apple", "orange" ) );
27    } // end main
28 } // end class MaximumTest
```