









# UPskill – JAVA + .NET

## Programação Orientada a Objetos – Interfaces Java

**Adaptado de Donald W. Smith (TechNeTrain)**

# Conteúdos

- Interfaces (“*interface types*”)
- Interface nativa “*Comparable*”
- Interface nativa “*Comparator*”

# *Interface Types*

- Uma interface é um tipo especial de declaração que lista um conjunto de métodos e suas assinaturas
  - Uma classe que implementa (*implements)* uma interface deve implementar todos os métodos da interface
  - Assemelha-se a uma classe, mas existem diferenças:
    - Todos os métodos numa interface são abstratos — têm um nome, parâmetros e um tipo de retorno, mas não têm uma implementação
    - Todos os métodos numa interface são automaticamente públicos
    - Uma interface não pode ter variáveis de instância
    - Uma interface não tem métodos estáticos

```
public interface Measurable
{
   double getMeasure();
}
```

Uma `interface` Java declara um conjunto de métodos e suas assinaturas

# *Interface*: Sintaxe de Declaração

- Declaração de uma interface e de uma classe que implementa (*implements*) a interface.

```
public interface Measurable
{
   double getMeasure();
}

public class BankAccount implements Measurable
{
   . . .

   public double getMeasure()
   {
      return balance;
   }
}
```

Interface methods are always public.

Interface methods have no implementation.

A class can implement one or more interface types.

Other BankAccount methods.

Implementation for the method that was declared in the interface type.

# Utilização das Interfaces

- Podemos usar a interface `Measurable` para implementar um método estático "universal" para cálculo de médias:

```
public interface Measurable
{
  double getMeasure();
}
```

```
public static double average(Measurable[] objs)
{
  if (objs.length == 0) return 0;
  double sum = 0;
  for (Measurable obj : objs)
  {
    sum = sum + obj.getMeasure();
  }
  return sum / objs.length;
}
```

# Implementação de uma *Interface*

- Uma classe pode ser declarada para `implementar` uma interface
  - A classe deve implementar todos os métodos da interface

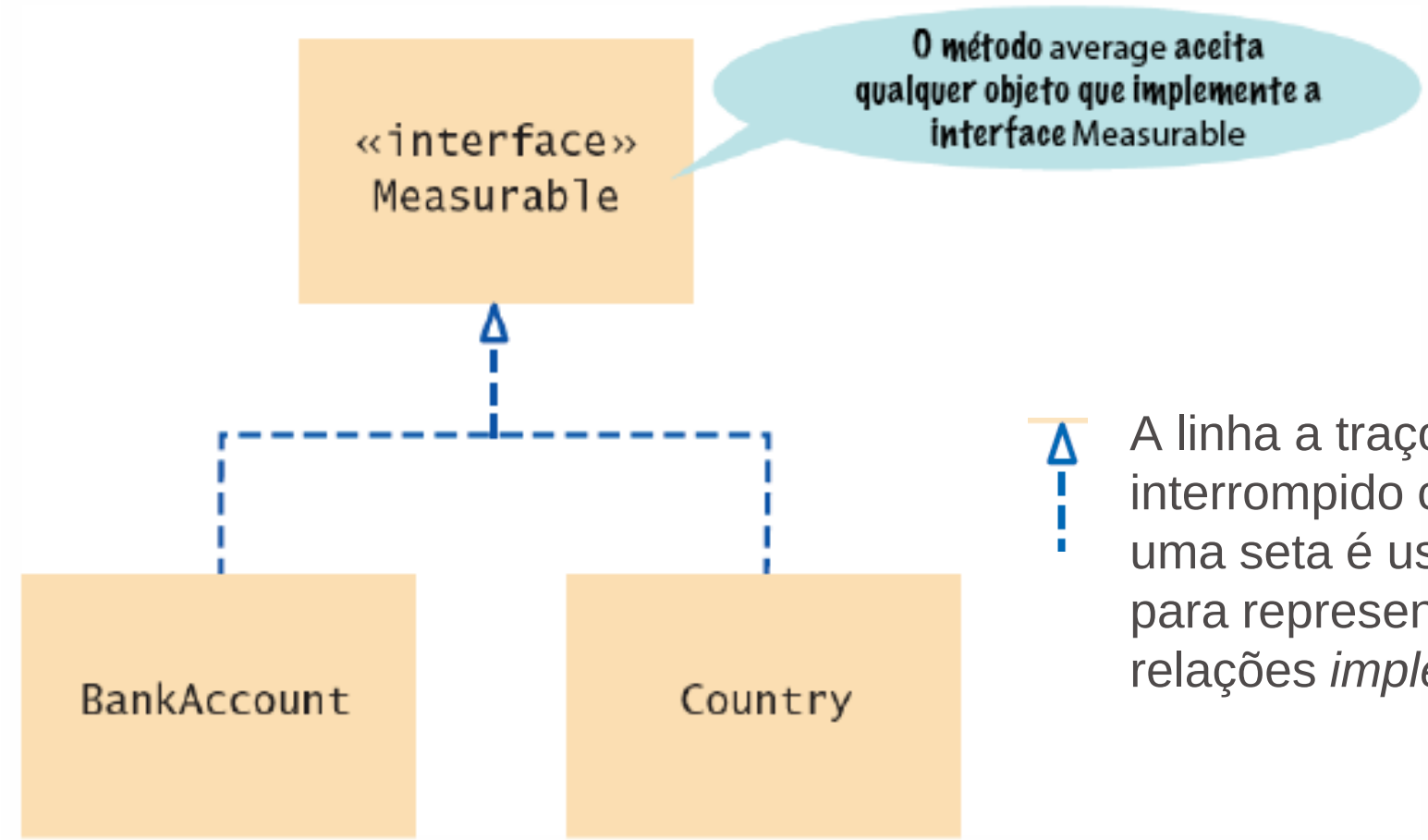
```
public class BankAccount implements Measurable
{
  public double getMeasure()
  {
    return balance;
  }
  ...
}
```

Usar a palavra reservada `implements` na declaração da classe

```
public class Country implements Measurable
{
  public double getMeasure()
  {
    return area;
  }
  ...
}
```

Os métodos da interface devem ser declarados como `public`

# Diagrama de Implementação




A linha a traço interrompido com uma seta é usada para representar relações *implements*

# MeasureableDemo.java (1)

UPskill
Digital Skills & Jobs

```
1  /**
2     This program demonstrates the measurable BankAccount and Country classes.
3  */
4  public class MeasurableDemo
5  {
6     public static void main(String[] args)
7     {
8        Measurable[] accounts = new Measurable[3];
9        accounts[0] = new BankAccount(0);
10       accounts[1] = new BankAccount(10000);
11       accounts[2] = new BankAccount(2000);
12
13       System.out.println("Average balance: "
14          + average(accounts));
15
16       Measurable[] countries = new Measurable[3];
17       countries[0] = new Country("Uruguay", 176220);
18       countries[1] = new Country("Thailand", 514000);
19       countries[2] = new Country("Belgium", 30510);
20
21       System.out.println("Average area: "
22          + average(countries));
23    }
```

# MeasureableDemo.java (2)

UPskill
Digital Skills & Jobs

```
25      /**
26         Computes the average of the measures of the given objects.
27         @param objs an array of Measurable objects
28         @return the average of the measures
29      */
30      public static double average(Measurable[] objs)
31      {
32         if (objs.length == 0) { return 0; }
33         double sum = 0;
34         for (Measurable obj : objs)
35         {
36            sum = sum + obj.getMeasure();
37         }
38         return sum / objs.length;
39      }
40   }
```

## Program Run

```
Average balance: 4000.0
Average area: 240243.33333333334
```

# Interface nativa *Comparable*

- A biblioteca Java inclui um conjunto de interfaces, entre elas a interface Comparable
  - Requer a implementação de um método: `compareTo()`
  - É usada para comparar dois objetos
  - É implementada por muitas classes da API do Java
  - Podemos implementá-la nas nossas classes para usar certas ferramentas da API Java, como a ordenação
- É invocada sobre um objeto e é passado outro objeto por parâmetro
  - Invocada sobre o objeto a, devolve os seguintes valores:
    - Negativo: a antecede b
    - Zero: a é igual a b
    - Positivo: a sucede b

```
a.compareTo(b);
```

# Parâmetro Comparable *Type*

- A interface `Comparable` usa um tipo especial de parâmetro que lhe permite lidar com qualquer tipo:

```
public interface Comparable<T>
{
  int compareTo(T other);
}
```

  - O tipo <T> é um espaço reservado para um tipo de objeto
  - A classe ArrayList usa a mesma técnica com o tipo envolto pelos símbolos < >

```
ArrayList<String> names = new ArrayList<String>();
```

# Exemplo *Comparable*

- O método `compareTo` da classe BankAccount compara contas bancárias através do seu saldo (*balance*)
  - Recebe um parâmetro do tipo da sua própria classe (BankAccount)

```
public class BankAccount implements Comparable<BankAccount>
{
  ...
  public int compareTo(BankAccount other)
  {
    if (balance < other.getBalance()) { return -1; }
    if (balance > other.getBalance()) { return 1; }
    return 0;
  }
  ...
}
```

Os métodos da interface devem ser declarados como public

# Utilização de *compareTo* para Ordenar

- O método `Arrays.sort` usa o método `compareTo` para ordenar os elementos do array
  - Uma vez que a classe `BankAccount` implementa a interface `Comparable`, podemos ordenar um vetor de contas bancárias com o método `Arrays.sort`.

```
BankAccount[] accounts = new BankAccount[3];
accounts[0] = new BankAccount(10000);
accounts[1] = new BankAccount(0);
accounts[2] = new BankAccount(2000);
Arrays.sort(accounts);
```

  - O vetor está agora ordenado por ordem crescente de saldo

A implementação das interfaces da biblioteca Java permite-nos usar as potencialidades da biblioteca com as nossas classes

# Erro Comum

- Esquecer de declarar a implementação dos métodos como public
  - Os métodos numa interface não são declarados como public, porque são public por omissão
  - Contudo, os métodos numa classe não são public por omissão
  - O esquecimento da palavra reservada public na declaração de um método de uma classe é frequente:

```
public class BankAccount implements Measurable
{
  double getMeasure()   // Oops—should be public
  {
    return balance;
  }
  ...
}
```

# Declaração de Constantes na Interface

- As interfaces não podem ter variáveis de instância, mas é possível declarar constantes
- Quando se declara uma constante numa interface, podemos (e devemos) omitir as palavras reservadas `public static final`, porque todas as constantes numa interface são automaticamente `public static final`

```
public interface SwingConstants
{
  int NORTH = 1;
  int NORTHEAST = 2;
  int EAST = 3;
  ...
}
```

# Interface nativa *Comparator*

- Esta interface não tem que ser implementada pelas classes cujos objetos se pretende comparar
- Uma outra classe pode implementar esta interface
- Usando esta interface, podemos implementar a ordenação baseada em diferentes atributos dos objetos a ordenar
  - Será necessário implementar uma ou mais classes responsáveis pela ordenação, de acordo com cada um dos critérios – estas classes implementam a interface `Comparator` e devem definir o método `compare()`

# Interface nativa *Comparator*

```
int compare(Object o1, Object o2)
```

- Este método compara os objetos o1 e o2 e devolve um inteiro:
  - Negativo: o1 antecede o2
  - Zero: o1 é igual a o2
  - Positivo: o1 sucede o2

# Exemplo *Comparator* (1)

- Implementação da ordenação de contas bancárias, primeiro por `owner (String)`, depois por `balance (double)`

```
public class BankAccount {
    private double balance;
    private String owner;

    public BankAccount(double bal, String name) {
        balance = bal;
        owner = name;
    }
    public double getBalance() {
        return balance;
    }
    public String getOwner() {
        return owner;
    }
    @Override
    public String toString() {
        return this.getClass().getName() + ":" + balance
    + ", " + owner;
    }
}
```

# Exemplo *Comparator* (2)

- Implementação da classe BankSortbyOwnerComparator que implementa a interface Comparator, definindo o método compare() para comparação de objetos da classe BankAccount através do atributo owner

```
import java.util.Comparator;

public class BankSortbyOwnerComparator implements Comparator<BankAccount> {
   @Override
   public int compare(BankAccount account1, BankAccount account2) {
        return account1.getOwner().compareTo(account2.getOwner());
   }
}
```

# Exemplo *Comparator* (3)

UPskill
Digital Skills & Jobs

**Objeto da classe que implementa a interface** Comparator

```
public class ComparatorMain {
   public static void main(String[] args) {
       BankAccount[] accounts = new BankAccount[3];
       accounts[0] = new BankAccount(1000, "Jose");
       accounts[1] = new BankAccount(2000, "Antonio");
       accounts[2] = new BankAccount(500, "Manuel");

        System.out.println("Before sort:");
        for (BankAccount account : accounts) {
            System.out.println(account);
        }
        Arrays.sort(accounts, new BankSortbyOwnerComparator());
        System.out.println("After sort by owner:");
        for (BankAccount account : accounts) {
            System.out.println(account);
        }
        …
```

# Exemplo *Comparator* (4)

UPskill
Digital Skills & Jobs

Classe anónima, sem nome, usada para criar um único objeto

```
public class ComparatorMain {
   // ...
      Arrays.sort(accounts, new Comparator<BankAccount>() {
          @Override
          public int compare(BankAccount account1, BankAccount account2) {
              return (account1.getBalance() < account2.getBalance() ) ? -1 :
                   (account1.getBalance() > account2.getBalance() ) ? 1 : 0 ;
          }
      }
      );
      System.out.println("After sort by balance:");
      for (BankAccount account : accounts) {
          System.out.println(account);
      }
   }
}
```

# Exemplo *Comparator* (5)

- Saída do programa

```
Before sort:
comparatormain.BankAccount:1000.0, Jose
comparatormain.BankAccount:2000.0, Antonio
comparatormain.BankAccount:500.0, Manuel
After sort by owner:
comparatormain.BankAccount:2000.0, Antonio
comparatormain.BankAccount:1000.0, Jose
comparatormain.BankAccount:500.0, Manuel
After sort by balance:
comparatormain.BankAccount:500.0, Manuel
comparatormain.BankAccount:1000.0, Jose
comparatormain.BankAccount:2000.0, Antonio
```

# Comparable vs Comparator

| | Comparable | Comparator |
|---|---|---|
| Lógica de ordenação | A lógica de ordenação deve estar na mesma classe cujos objetos são ordenados – ordenação natural | A lógica de ordenação encontra-se numa classe separada – permite criar diferentes ordenações baseadas em diferentes atributos |
| Implementação | A classe cujos objetos são ordenados deve implementar a interface | A classe cujos objetos são ordenados não implementa a interface – uma outra classe que implementa a interface é usada para definir o método de comparação |
| Método | `int compareTo(Object o1)`<br>Este método compara o objeto `this` com `o1` | `int compare(Object o1, Object o2)`<br>Este método compara os objetos `o1` e `o2` |
| Package | `java.lang.Comparable` | `java.util.Comparator` |

# Resumo

- O tipo `interface` contém os tipos de retorno, nomes e parâmetros de um conjunto de métodos
- Ao contrário de uma classe, uma `interface` não dispõe de implementação
- Um método que tenha um tipo interface como parâmetro, pode aceitar objetos de diferentes classes
- O termo `implements` indica quais são as interfaces que uma classe implementa
- Os objetos de uma classe que implemente a interface `Comparable`, podem ser comparados num método de ordenação