Classes Abstratas e Interfaces

Programação Orientada a Objetos — QXD0007

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará

2º semestre/2021

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Leituras para esta aula

- **Capítulos 10 e 11** (Classes Abstratas e Interfaces) da apostila da Caelum – Curso FJ-11, Disponível no link: https://www.caelum.com.br/apostila/apostila-java-orientacao-objetos.pdf

- **Capítulo 10** (Polimorfismo e Interfaces) do livro Java Como Programar, Décima Edição, Disponível no link: http://libgen.lc/ads.php?md5=728636A04ACA056038BB5F079403AC96

- **Capítulo 9** (Classes abstratas e interfaces) do livro Introdução à Programação Orientada a Objetos usando Java, Rafael Santos, Disponível no link: https://www.academia.edu/6227746/Introdu%C3%A7%C3%A3o_%C3%A0_Programa%C3%A7%C3%A3o_Orientada_a_Objetos_Usando_Java

Introdução

Motivação

- Na herança, devemos criar uma classe ancestral que tenha os campos e métodos comuns a todas as suas herdeiras, e devemos fazer a implementação dos métodos de forma que instâncias da classe ancestral possam ser criadas.

Motivação

- Na herança, devemos criar uma classe ancestral que tenha os campos e métodos comuns a todas as suas herdeiras, e devemos fazer a implementação dos métodos de forma que instâncias da classe ancestral possam ser criadas.

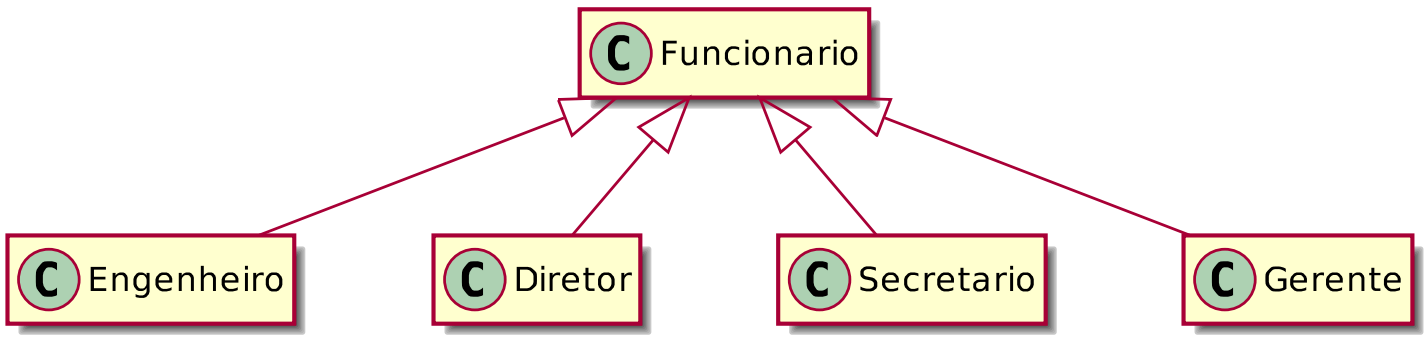
- **Nem sempre isto é desejável!**

- Em alguns casos seria interessante declarar os campos e métodos que as classes herdeiras devem implementar, mas não permitir a criação de instâncias da classe ancestral.

- Desta forma, a classe ancestral passaria a ser somente um guia de que métodos e campos deveriam ser implementados nas classes herdeiras. A classe ancestral ditaria para as classes descendentes o que deve ser feito, mas sem necessariamente dizer como deve ser feito.

Cenário 1: Funcionários de uma empresa

- No sistema de uma empresa, a superclasse `Funcionario` encapsula os atributos e métodos comuns a todos os empregados e os atributos e métodos específicos de cada empregado são declarados e implementados nas respectivas subclasses.



Exemplo de um pedaço do código deste sistema

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```java
public class Funcionario {

    protected String nome;
    protected String cpf;
    protected double salario;

    public double getBonificacao() {
        return this.salario * 1.2;
    }

    // outros métodos aqui

}
```

$\Leftarrow$ Classe Funcionario

Exemplo de um pedaço do código deste sistema

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```java
public class Funcionario {

    protected String nome;
    protected String cpf;
    protected double salario;

    public double getBonificacao() {
        return this.salario * 1.2;
    }

    // outros métodos aqui

}
```

⇐ Classe Funcionario

```java
public class ControleDeBonificacoes {

    private double totalDeBonificacoes = 0;

        public void registra(Funcionario f) {
            System.out.println("Adicionando bonificação do funcionario: " + f);
            this.totalDeBonificacoes += f.getBonificacao();
        }

        public double getTotalDeBonificacoes() {
            return this.totalDeBonificacoes;
        }
}
```

⇐ Controle de bonificações. Observe o polimorfismo.

Observações sobre essa hierarquia de herança

- **Não precisamos ter um objeto Funcionário no sistema**
- Estamos usando a classe `Funcionario` para o polimorfismo.
- Se não fosse ela, precisaríamos criar um método `registra` para cada um dos tipos de `Funcionario`, um para `Gerente`, outro para `Engenheiro`, `Diretor`, etc.

Observações sobre essa hierarquia de herança

- **Não precisamos ter um objeto Funcionário no sistema**
- Estamos usando a classe `Funcionario` para o polimorfismo.
- Se não fosse ela, precisaríamos criar um método `registra` para cada um dos tipos de `Funcionario`, um para `Gerente`, outro para `Engenheiro`, `Diretor`, etc.
- Em alguns sistemas, como é neste caso, usamos uma classe com apenas esse intuito: economizar um pouco de código e ganhar polimorfismo para criar métodos mais genéricos, que se encaixem a diversos objetos.

Observações sobre essa hierarquia de herança

- **Não precisamos ter um objeto Funcionário no sistema**
- Estamos usando a classe `Funcionario` para o polimorfismo.
- Se não fosse ela, precisaríamos criar um método `registra` para cada um dos tipos de `Funcionario`, um para `Gerente`, outro para `Engenheiro`, `Diretor`, etc.
- Em alguns sistemas, como é neste caso, usamos uma classe com apenas esse intuito: economizar um pouco de código e ganhar polimorfismo para criar métodos mais genéricos, que se encaixem a diversos objetos.
- **Em algumas aplicações, precisamos de referências para superclasses genéricas como a Funcionário**
  Para obtermos o polimorfismo.

Cenário 2: Pessoa física e Pessoa jurídica

- Suponha que em um negócio relacionado a banco, apenas Pessoa física e Pessoa jurídica são permitidas.

- Imagine a superclasse `Pessoa` e duas filhas: `PessoaFisica` e `PessoaJuridica`.

Cenário 2: Pessoa física e Pessoa jurídica

- Suponha que em um negócio relacionado a banco, apenas Pessoa física e Pessoa jurídica são permitidas.

- Imagine a superclasse `Pessoa` e duas filhas: `PessoaFisica` e `PessoaJuridica`.

- Quando puxamos relatórios de nossos clientes (um ArrayList de `Pessoa`, por exemplo), queremos que cada um deles seja ou uma `PessoaFisica` ou uma `PessoaJuridica`.

Cenário 2: Pessoa física e Pessoa jurídica

- Suponha que em um negócio relacionado a banco, apenas Pessoa física e Pessoa jurídica são permitidas.

- Imagine a superclasse `Pessoa` e duas filhas: `PessoaFisica` e `PessoaJuridica`.

- Quando puxamos relatórios de nossos clientes (um ArrayList de `Pessoa`, por exemplo), queremos que cada um deles seja ou uma `PessoaFisica` ou uma `PessoaJuridica`.

- A classe `Pessoa`, neste caso, estaria sendo usada apenas para ganhar o polimorfismo e herdar algumas coisas:
  não faz sentido permitir instanciá-la.

Questionamento

- Se a classe `Pessoa` não pode ser instanciada, por que simplesmente não criar somente `PessoaFisica` e `PessoaJuridica`?

- Resposta:
  - **Reuso**
  - **Polimorfismo**: a superclasse classe genérica nos permite tratar de forma fácil e uniforme todos os tipos de pessoa, inclusive com polimorfismo se for o caso. Por exemplo, você pode colocar todos tipos de contas em uma mesma coleção.

Classes abstratas e Interfaces

- A linguagem Java tem dois mecanismos que permitem a criação de classes que somente contêm descrições de atributos e métodos que devem ser implementados, mas sem efetivamente implementar os métodos:
  - Classes abstratas
  - Interfaces

Classes abstratas

Classe abstrata

Uma classe abstrata é declarada com o modificador `abstract`.

Classes abstrata são classes que não podem ser instanciadas, mas podem ser herdadas.

É uma forma de garantir herança total: somente subclasses não abstratas podem ser instanciadas, mas nunca a superclasse abstrata

Classe Abstrata

- Lembrando do Cenário 1, era inadmissível ter um objeto que fosse apenas do tipo `Funcionario`.

Classe Abstrata

- Lembrando do Cenário 1, era inadmissível ter um objeto que fosse apenas do tipo `Funcionario`.

- Em Java, a fim de impedir que uma classe seja instanciada, declaramos ela como uma classe abstrata por meio do modificador `abstract`.

```
public abstract class Funcionario {

    protected double salario;

    public double getBonificacao() {
        return this.salario * 1.2;
    }

    // outros atributos e métodos comuns a todos Funcionarios

}
```

Classe Abstrata

- Uma classe abstrata não pode ser instanciada. Qualquer tentativa de instanciação gera um erro de compilação.
  - Analisar os arquivos `Funcionario.java` e `FuncionarioTeste.java`

Classe Abstrata

- Uma classe abstrata não pode ser instanciada. Qualquer tentativa de instanciação gera um erro de compilação.
  - Analisar os arquivos `Funcionario.java` e `FuncionarioTeste.java`

- Se uma classe abstrata não pode ser instanciada, então para quê ela serve?
  Serve para o polimorfismo e herança dos atributos e métodos

Relembrando o Cenário 1


```
public double getBonificacao() {
    return this.salario * 1.2;
}
```

- Se o método `getBonificacao` não for reescrito na subclasse, ele será herdado da superclasse, fazendo com que seja devolvido o salário mais 20%

Relembrando o Cenário 1


```java
public double getBonificacao() {
    return this.salario * 1.2;
}
```

- Se o método `getBonificacao` não for reescrito na subclasse, ele será herdado da superclasse, fazendo com que seja devolvido o salário mais 20%

- No entanto, não existe uma bonificação padrão para todo tipo de `Funcionario`.
  - Levando em consideração que cada funcionário no sistema tem uma regra totalmente diferente de ser bonificado, não faz sentido que este método seja implementado na classe `Funcionario`.

Relembrando o Cenário 1

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```java
public double getBonificacao() {
    return this.salario * 1.2;
}
```

- Se o método `getBonificacao` não for reescrito na subclasse, ele será herdado da superclasse, fazendo com que seja devolvido o salário mais 20%

- No entanto, não existe uma bonificação padrão para todo tipo de `Funcionario`.
  - Levando em consideração que cada funcionário no sistema tem uma regra totalmente diferente de ser bonificado, não faz sentido que este método seja implementado na classe `Funcionario`.

> Queremos que cada pessoa que escreve uma subclasse de `Funcionario` sobrescreva o método `getBonificacao` de acordo com suas regras.

Métodos Abstratos

Método abstrato é um método que é declarado com o modificador `abstract` e é declarado sem uma implementação.

Exemplo: `abstract void moveTo(double X, double Y);`

Métodos Abstratos

Método abstrato é um método que é declarado com o modificador `abstract` e é declarado sem uma implementação.

Exemplo: `abstract void moveTo(double X, double Y);`

- Métodos abstratos são declarados com o modificador `abstract`.
- Se uma classe contém um método abstrato, suas subclasses deverão **obrigatoriamente** implementar o método abstrato com mesmo nome, modificador, tipo de retorno e argumentos declarados na superclasse.

Classes Abstratas II

- Se uma classe tiver métodos abstratos, ela também deverá **obrigatoriamente** ser declarada como `abstract`.

- Uma subclasse de uma superclasse abstrata deve, **obrigatoriamente**, implementar todos os métodos abstratos da superclasse, se houver algum. Caso não haja, nenhuma implementação é obrigatória.

- Se uma subclasse de uma classe abstrata deixar de implementar algum método abstrato que estiver na superclasse, **automaticamente** a subclasse torna-se abstrata e deve ser declarada com o modificador `abstract`.

- Classes abstratas podem ter atributos e podem implementar alguns métodos (implementação parcial).

Classes Abstratas II

- Construtores de classes abstratas não podem ser `abstract`.
  - Mesmo que a classe abstrata não possa ser instanciada, seus construtores podem inicializar os campos da classe que serão usados por subclasses, sendo imprescindíveis em praticamente todos os casos.

- Uma classe abstrata pode ter métodos estáticos, contanto que eles não sejam abstratos. Ela também pode ter atributos estáticos. O funcionamento desses atributos e métodos estáticos é igual ao que já conhecemos.

Exemplo: modelando robôs

- Vejamos o exemplo de classes que implementam robôs para simulação.

- Podemos considerar que existem mecanismos básicos que são comuns a todos os tipos de robôs, mas sua implementação será diferente dependendo do tipo de robô.

- Exemplo: movimento do robô.
  - Um robô simples pode se movimentar somente nas quatro posições cardeais, um robô mais complexo em várias direções angulares, um robô movido a energia limitada pode contabilizar quanta energia foi gasta no movimento, etc.

  - Cada um desses mecanismos deveria ser implementado de maneira diferente, e não necessariamente tendo algo em comum.

Exemplo: modelando robôs

- Nesse caso, poderíamos escrever uma superclasse contendo a declaração do método de movimento, sem necessariamente conter os comandos para implementá-lo

- Analisar o `ProjetoRobo`.

Interfaces

Interface

- Vimos que classes abstratas podem conter métodos não-abstratos.
- Se a classe tiver apenas métodos abstratos, podemos criá-la como uma interface.

Interface

- Vimos que classes abstratas podem conter métodos não-abstratos.
- Se a classe tiver apenas métodos abstratos, podemos criá-la como uma interface.

Uma interface é um template de classe.

Assim como uma classe abstrata, uma interface não pode ser instanciada. Todos os métodos na interface são *implicitamente* `abstract` e `public`, e não podem ser declarados com seus corpos.

Se houverem atributos, estes serão *implicitamente* considerados `static` e `final`, devendo, portanto, ser inicializados na sua declaração.

Definindo uma interface

Uma interface é definida através da palavra-chave `interface`.

```
1 public interface Conta {
2     void depositar (double valor);
3     void sacar (double valor);
4     double getSaldo();
5 }
```

Para uma classe implementar uma interface, é usada a palavra-chave `implements`.

Definindo uma interface

Uma interface é definida através da palavra-chave `interface`.

```
1 public interface Conta {
2     void depositar (double valor);
3     void sacar (double valor);
4     double getSaldo();
5 }
```

Para uma classe implementar uma interface, é usada a palavra-chave `implements`.

Analisar o `ProjetoConta`

Interface vs. Classe Abstrata

- Em Java, uma subclasse somente pode herdar de uma única superclasse (abstrata ou não).

- Porém, qualquer classe em Java pode implementar **várias** interfaces simultaneamente.
  - Interfaces são, então, um mecanismo simplificado de herança múltipla em Java, permitindo que mais de uma interface determine os métodos que uma classe herdeira deve implementar.

Interface vs. Classe Abstrata

- Em Java, uma subclasse somente pode herdar de uma única superclasse (abstrata ou não).

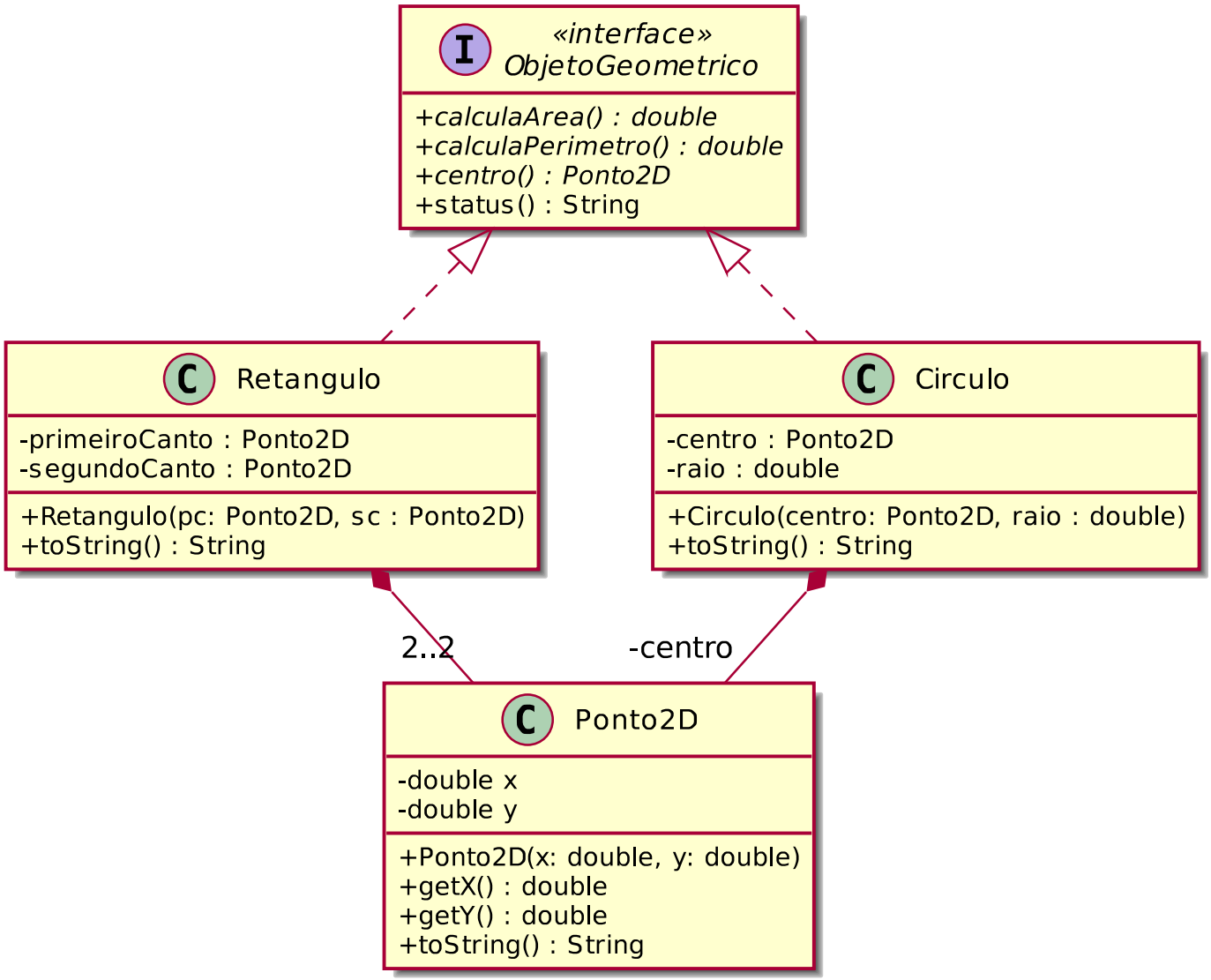
- Porém, qualquer classe em Java pode implementar **várias** interfaces simultaneamente.
  - Interfaces são, então, um mecanismo simplificado de herança múltipla em Java, permitindo que mais de uma interface determine os métodos que uma classe herdeira deve implementar.

- **Interfaces como bibliotecas de constantes:** já que todos os atributos de uma interface são declarados como `static` e `final`, podemos escrever interfaces que somente contêm atributos, e qualquer classe que implementar essa interface terá acesso a estas constantes.

Interfaces

- A partir do Java SE 8, é possível fornecer uma implementação *default* para qualquer método de uma interface.
- Se o método tiver uma implementação, antes da sua definição deve ser colocada a palavra-chave `default`.
- Vamos ilustrar isso no exemplo a seguir.

Interface – Mais um exemplo



- Analisar o Projeto `ObjetoGeometrico`

Herança múltipla usando Interfaces

Herança múltipla usando Interfaces

- A principal diferença entre herança usando classes abstratas e usando interfaces é que uma classe pode herdar somente de uma única classe, enquanto pode implementar diversas interfaces.

- Um exemplo desse mecanismo é mostrado a seguir:

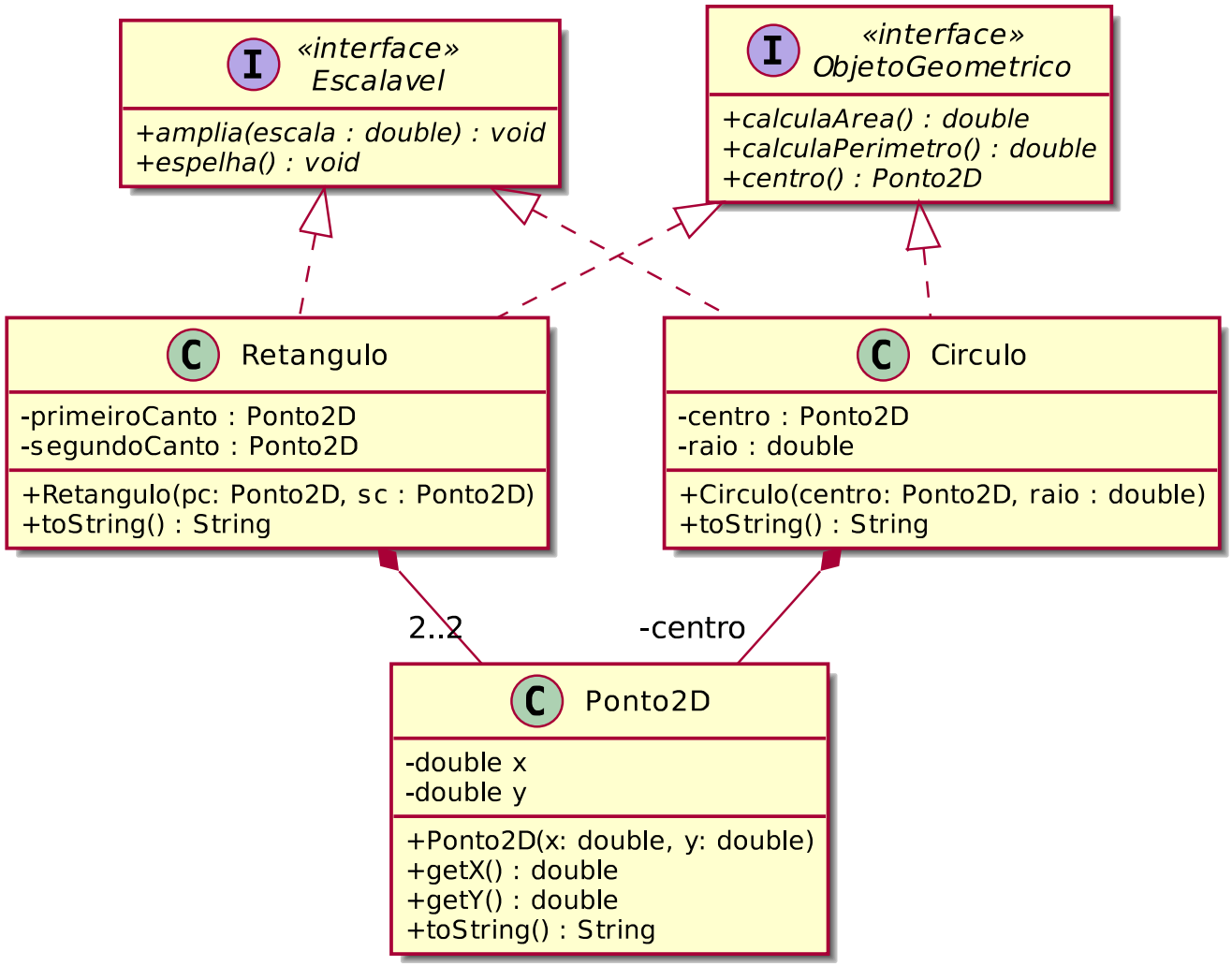
Herança múltipla usando Interfaces

- A principal diferença entre herança usando classes abstratas e usando interfaces é que uma classe pode herdar somente de uma única classe, enquanto pode implementar diversas interfaces.

- Um exemplo desse mecanismo é mostrado a seguir:

- **Exemplo:** Considere que os objetos geométricos tratados no exemplo do slide anterior possam ser escaláveis, isto é, o seu tamanho original pode ser modificado usando-se um valor como escala. Os dados que representam o tamanho do objeto seriam modificados por um método que recebesse a escala como argumento.

Herança múltipla usando Interfaces



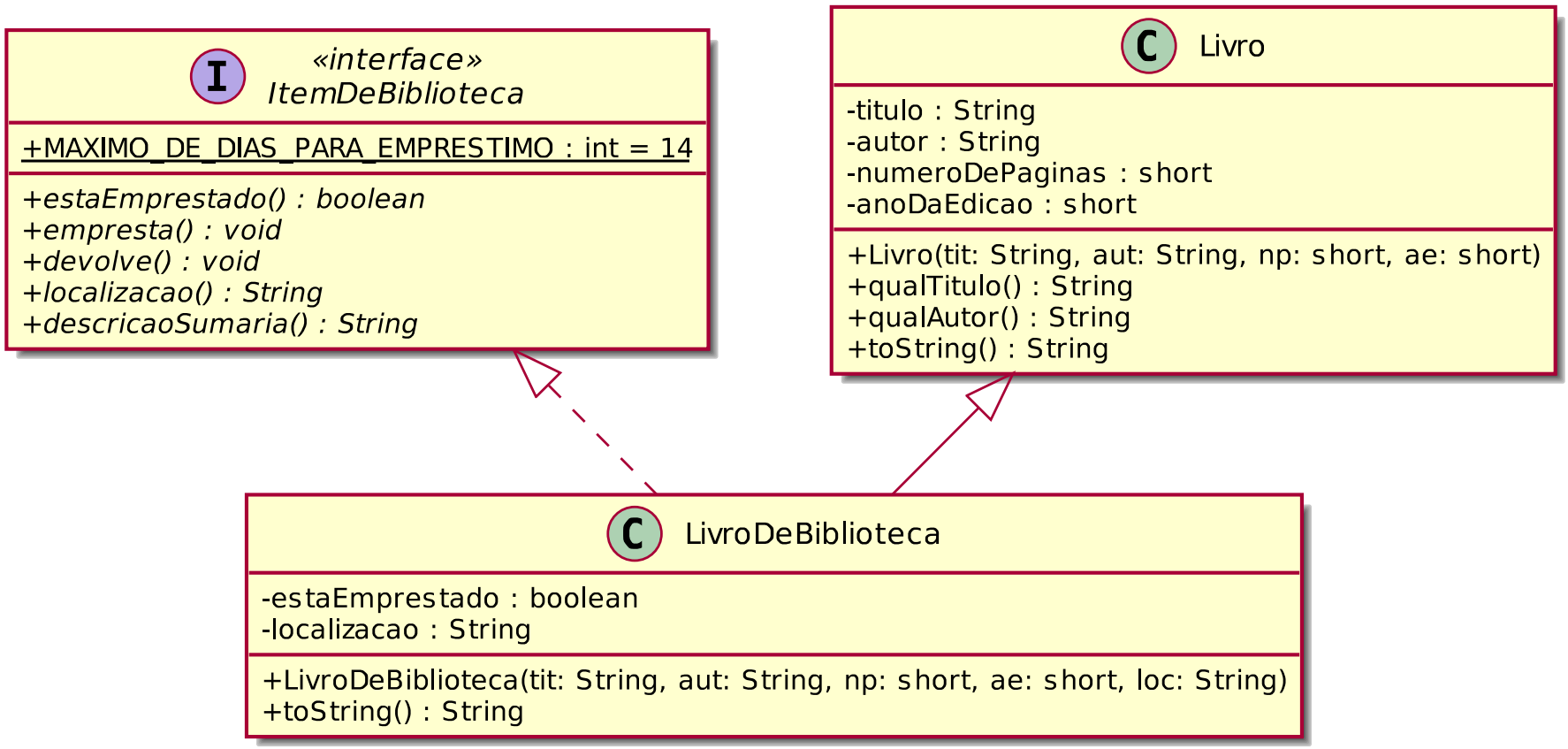
Analisar o Projeto `ObjetosEscalaveis`

Herança múltipla usando interfaces

- Também é possível implementar herança múltipla em Java fazendo com que uma classe herde de outra mas implemente uma ou mais interfaces.

- Um exemplo de herança múltipla de classes e interfaces é dada a seguir, onde consideramos uma hierarquia de classes e interfaces que representam itens de uma biblioteca.

Herança múltipla usando interfaces



Analisar ProjetoBiblioteca.

Conflitos de Herança Múltipla

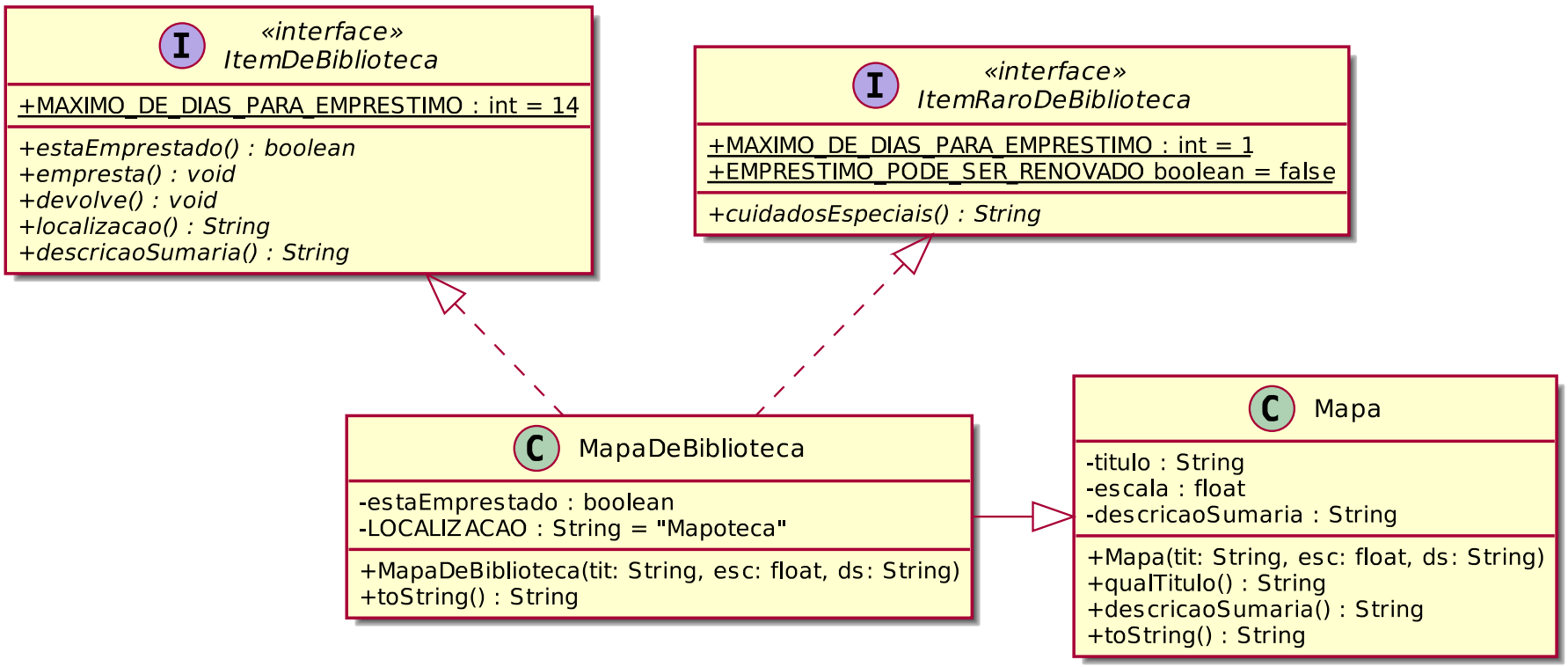
Conflitos de Herança Múltipla

- **Problema que pode ocorrer com herança múltipla:** uma classe deve implementar mais de uma interface, e duas ou mais interfaces declaram campos com o mesmo nome – a classe que implementa os métodos não poderá ser compilada por causa de um conflito de nomes.

- A fim de exemplificar esse caso, modificamos o exemplo de itens de uma biblioteca dado no slide anterior para considerar que alguns itens de biblioteca são raros e devem ter cuidados adicionais quando forem emprestados, além de ter um prazo de empréstimo diferente.

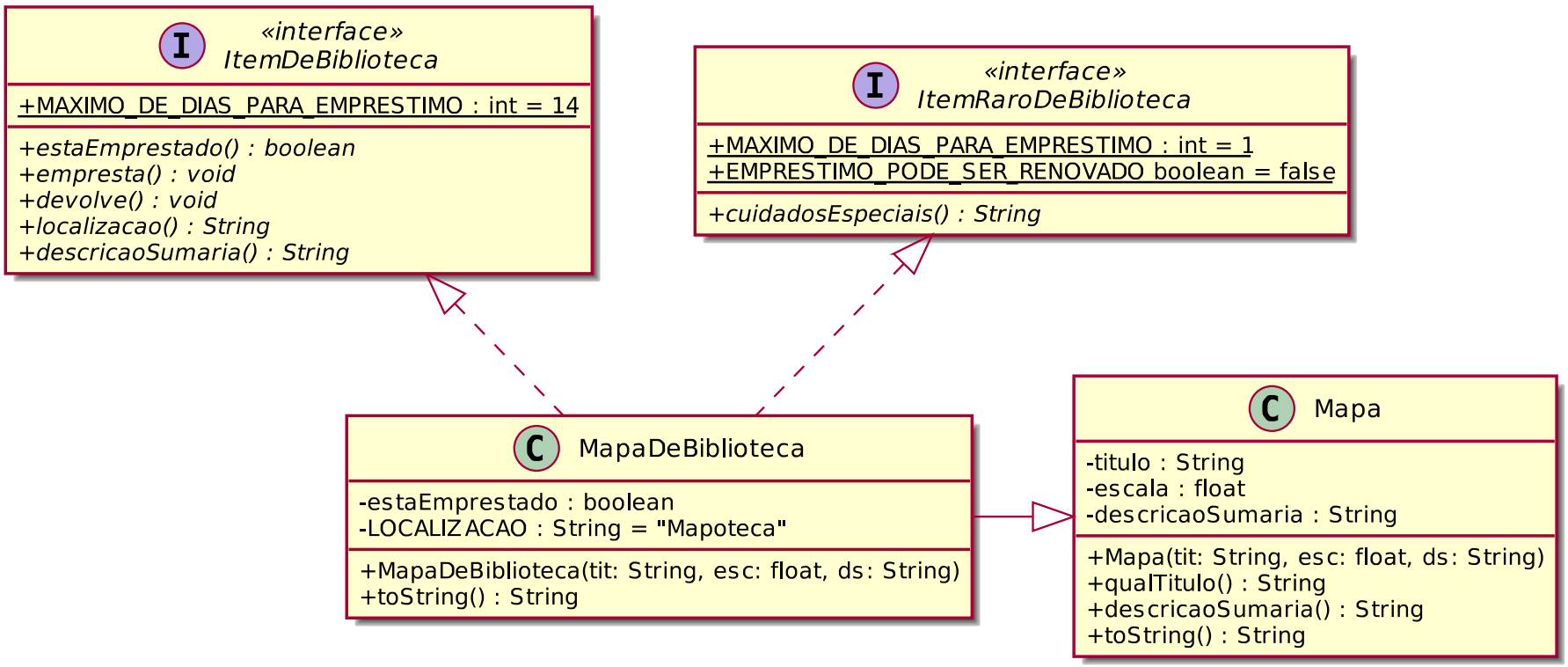
- Para definir métodos em classes que encapsulam itens raros de biblioteca, criamos uma nova interface `ItemRaroDeBiblioteca`. Também criamos as classes `Mapa` e `MapaDeBiblioteca`. Geralmente, mapas de bibliotecas são itens raros.

Conflitos de Herança Múltipla



Analisar ProjetoConflito.

Conflitos de Herança Múltipla



Analisar ProjetoConflito.

Solução: Para resolver o conflito, é preciso indicar, na classe `MapaDeBiblioteca`, que a constante `MAXIMO_DE_DIAS_PARA_EMPRESTIMO` a ser considerada pertence à interface `ItemRaroDeBiblioteca`.

Conflitos de Herança Múltipla

- Se duas interfaces possuem o mesmo método (a mesma assinatura), e elas não fornecem nenhuma implementação para o método, então não há conflito nenhum.

- A classe que implementar as duas interfaces pode decidir entre implementar o método abstrato ou não. Caso ela implemente, o método servirá tanto a uma quanto à outra interface; agora, se a classe não implementar o método, então ela se tornará abstrata.

Conflitos de Herança Múltipla

- O que acontece se um mesmo método é definido como `default` em uma interface e é definido também em uma superclasse ou outra interface?

Conflitos de Herança Múltipla

- O que acontece se um mesmo método é definido como `default` em uma interface e é definido também em uma superclasse ou outra interface?

- Java tem regras simples para resolver este conflito:
  1) Superclasses ganham. Se a superclasse fornece uma implementação do método, métodos default com a mesma assinatura são ignorados.
  2) Interfaces conflitam. Se uma interface fornece um método default e outra interface possui um método (default ou não) com a mesma assinatura, então você deve resolver o conflito sobrepondo o método conflitante.

Usando a interface `Comparable` do Java

A interface `Comparable` do Java

- A interface `Comparable` é usada para permitir que uma colletion ordene objetos de uma classe definida pelo usuário com base em sua ordem natural.

A interface `Comparable` do Java

- A interface `Comparable` é usada para permitir que uma colletion ordene objetos de uma classe definida pelo usuário com base em sua ordem natural.

- Essa interface é encontrada no pacote `java.lang` e contém apenas um método chamado `compareTo`

```
public interface Comparable<T>
{
   int compareTo(T other); // parameter has type T
}
```

- O tipo `T` é um "tipo genérico" que deve ser substituído pelo nome da classe que implementa esta interface.

A interface `Comparable` do Java

- A interface `Comparable` é usada para permitir que uma colletion ordene objetos de uma classe definida pelo usuário com base em sua ordem natural.
- Essa interface é encontrada no pacote `java.lang` e contém apenas um método chamado `compareTo`

```
public interface Comparable<T>
{
   int compareTo(T other); // parameter has type T
}
```

- O tipo `T` é um "tipo genérico" que deve ser substituído pelo nome da classe que implementa esta interface.
- A ordenação natural de elementos é imposta pela implementação do método `compareTo()`.

A interface Comparable do Java

- A interface `Comparable` é usada para permitir que uma colletion ordene objetos de uma classe definida pelo usuário com base em sua ordem natural.
- Essa interface é encontrada no pacote `java.lang` e contém apenas um método chamado `compareTo`

```
public interface Comparable<T>
{
   int compareTo(T other); // parameter has type T
}
```

- O tipo `T` é um "tipo genérico" que deve ser substituído pelo nome da classe que implementa esta interface.
- A ordenação natural de elementos é imposta pela implementação do método `compareTo()`.

Isso significa que qualquer classe que implemente a interface `Comparable` é obrigada a ter uma implementação do método `compareTo`.

A interface `Comparable`

- Para que qualquer classe suporte a ordenação natural, ela deve implementar a interface `Comparable` e implementar o método `compareTo()`. Este método recebe um objeto como argumento e retorna um inteiro.

A interface `Comparable`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Para que qualquer classe suporte a ordenação natural, ela deve implementar a interface `Comparable` e implementar o método `compareTo()`. Este método recebe um objeto como argumento e retorna um inteiro.

- `int compareTo (T obj)`: É usado para comparar o objeto atual com o objeto passado como argumento. Devolve:
  - **inteiro negativo**, se o objeto atual for menor que o argumento.
  - **zero**, se o objeto atual for igual ao argumento.
  - **inteiro positivo**, se o objeto atual for maior que o argumento.

Exemplo

- Temos uma classe `Pessoa` que possui apenas dois atributos: `id` e `nome`.
- Gostaríamos de comparar duas pessoas pelo `id` delas.

Exemplo

- Temos uma classe `Pessoa` que possui apenas dois atributos: `id` e `nome`.
- Gostaríamos de comparar duas pessoas pelo `id` delas.
- Para isso, a classe `Pessoa` deve implementar a interface `Comparable`

```
public class Pessoa implements Comparable<Pessoa> {...}
```

Exemplo

- Temos uma classe `Pessoa` que possui apenas dois atributos: `id` e `nome`.
- Gostaríamos de comparar duas pessoas pelo `id` delas.
- Para isso, a classe `Pessoa` deve implementar a interface `Comparable`

```
public class Pessoa implements Comparable<Pessoa> {...}
```

- Como `Pessoa` implementa esta interface, ela deve obrigatoriamente fornecer uma implementação para o método `compareTo`:

```
public int compareTo(Pessoa p) {
    return Integer.compare(this.id, p.getId());
}
```

Pessoa.java


```java
public class Pessoa implements Comparable<Pessoa> {
    private int id;
    private String nome;

    public Pessoa(int id, String nome) {
        this.id = id;
        this.nome = nome;
    }

    public int getId() { return id; }
    public String getNome() { return nome; }

    @Override public int compareTo(Pessoa p) {
        return Integer.compare(this.id, p.getId());
    }

    @Override public String toString() {
        return getClass().getName() +
            "[" + id + ":" + nome + "]";
    }
}
```

E agora?

- Suponha que exista um método de ordenação que promete ordenar listas de objetos, contanto que eles implementem a interface `Comparable`.

- Agora, podemos passar uma lista de objetos do tipo `Pessoa` para esse método e ele ordenará nossos objetos de acordo com o `id` deles!!

Ordenando arrays de objetos

- Como já vimos, a classe **java.util.Arrays** contém vários métodos que permitem a manipulação e o processamento de arrays.

Ordenando arrays de objetos

- Como já vimos, a classe **java.util.Arrays** contém vários métodos que permitem a manipulação e o processamento de arrays.

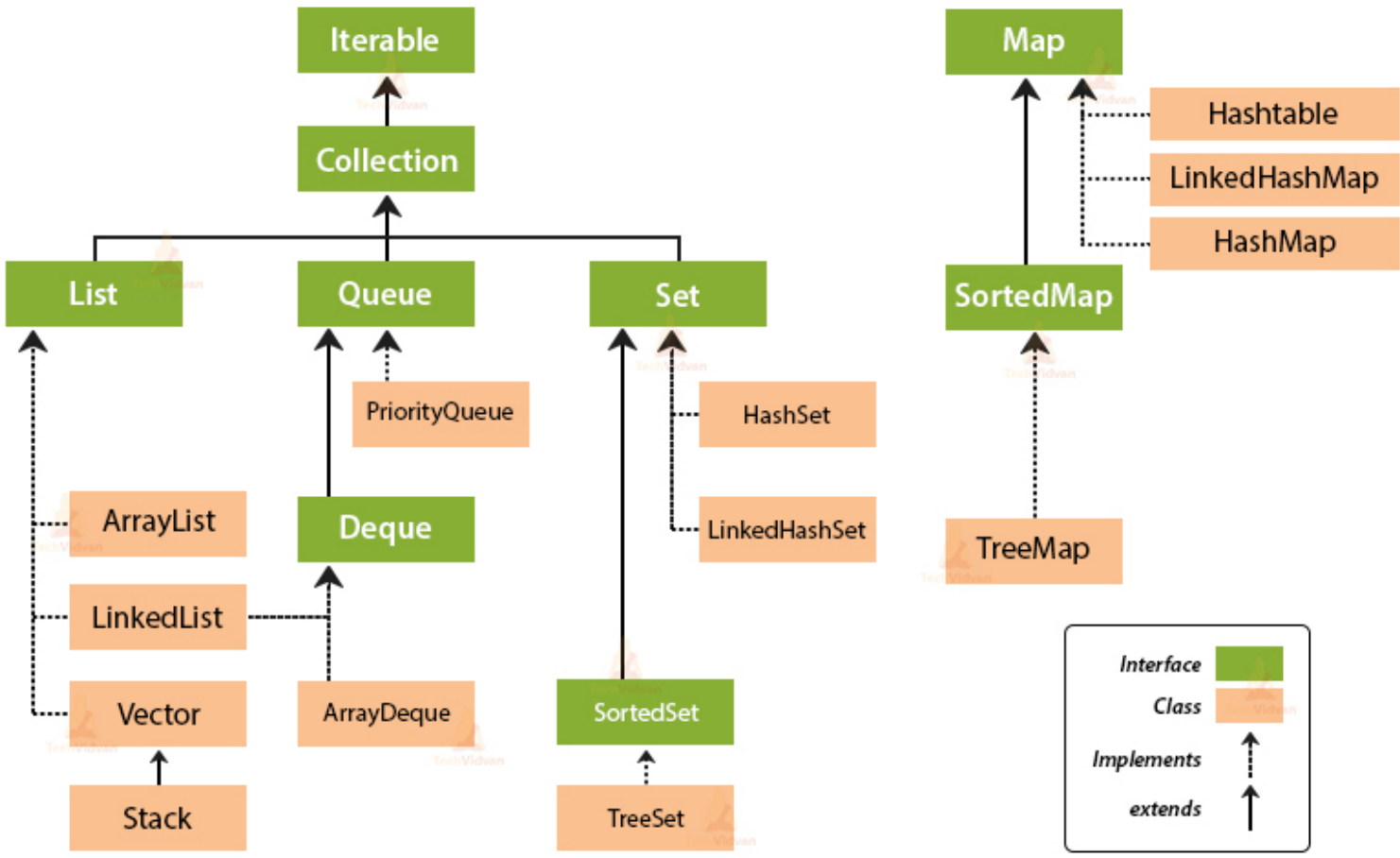
- Por exemplo, o método `sort(T arr[])` da classe `Arrays` promete ordenar um vetor de objetos sob uma condição:
  - Os objetos contidos no vetor **devem** pertencer a uma classe que implemente a interface `Comparable`.

Ordenando arrays de objetos

- Como já vimos, a classe **java.util.Arrays** contém vários métodos que permitem a manipulação e o processamento de arrays.

- Por exemplo, o método `sort(T arr[])` da classe `Arrays` promete ordenar um vetor de objetos sob uma condição:
  - Os objetos contidos no vetor **devem** pertencer a uma classe que implemente a interface `Comparable`.

- Exemplo: Suponha que temos um vetor com objetos do tipo `Aluno` e gostaríamos de usar o método `Arrays.sort()` da classe `Arrays` para ordenar esse vetor. Para tornar isso possível basta modificar a classe para que ela implemente a interface `Comparable`.



- `ArrayList` e `LinkedList` estendem a interface `List` do Java.

O método `Collections.sort()`

- O Java também possui o método estático `sort()` que pertence à classe `Collections` do pacote `java.util`

- Este método pode ordenar diversas coleções de objetos como, por exemplo, ArrayLists e LinkedLists.

- Porém, se os objetos da coleção pertenceram a uma classe definida pelo usuário, o método `sort` não funcionará adequadamente, pois ele não sabe automaticamente como comparar dois objetos definidos pelo usuário. Desta forma, o usuário tem que "ensiná-lo" como comparar.
  - Uma das formas de fazer isso é sua classe implementar a interface `Comparable`.
  - Com o método `compareTo` definido na sua classe, agora o método `sort` saberá como comparar dois objetos da sua classe e poderá ordenar corretamente a usa ArrayList.

Exemplo

- Analisar o Projeto SortingArrays

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM