Classes Abstratas e Interfaces

Programação Orientada a Objetos — QXD0007

Prof. Atílio Gomes Luiz

gomes.atilio@ufc.br

Universidade Federal do Ceará

1º semestre/2022

Leituras para esta aula

- **Capítulos 10 e 11** (Classes Abstratas e Interfaces) da apostila da Caelum – Curso FJ-11

- **Capítulo 10** (Polimorfismo e Interfaces) do livro Java Como Programar, Décima Edição

Introdução

Motivação

- Na herança, devemos criar uma classe ancestral que tenha os campos e métodos comuns a todas as suas herdeiras, e devemos implementar os métodos para que instâncias da classe ancestral possam ser criadas.

Motivação

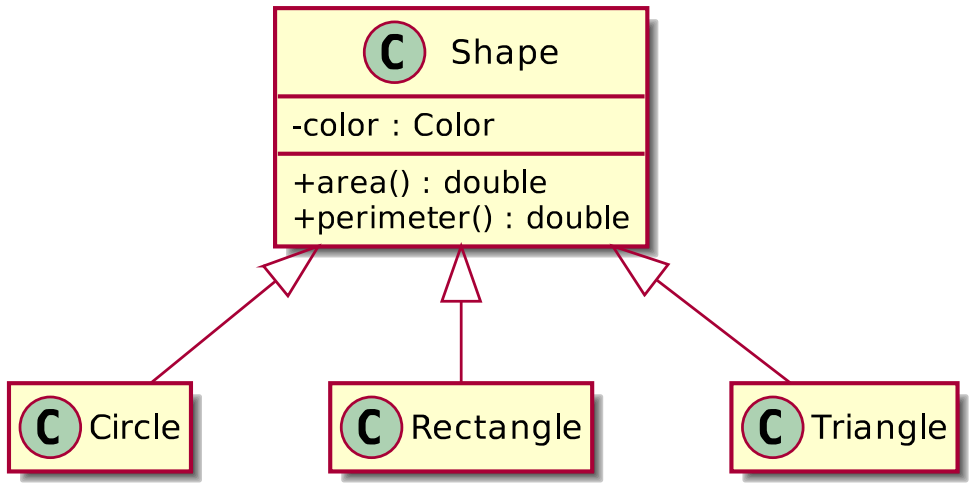
- Na herança, devemos criar uma classe ancestral que tenha os campos e métodos comuns a todas as suas herdeiras, e devemos implementar os métodos para que instâncias da classe ancestral possam ser criadas.

- **Nem sempre isto é desejável!**

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

Cenário 1: Formas Geométricas

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



- Queremos guardar uma lista de formas geométricas (círculos, retângulos ou triângulos) para depois imprimir a área e o perímetro delas.

- As formas geométricas podem ter uma cor associada.

Cenário 1: Formas Geométricas

No programa-cliente teríamos os seguintes métodos estáticos:

```
1  public static void printAreas(List<Shape> list)
2  {
3      for(Shape figure : list)
4          System.out.println(figure.area());
5  }
6
7  public static void printPerimeters(List<Shape> list)
8  {
9      for(Shape figure : list)
10         System.out.println(figure.perimeter());
11 }
```

Observações sobre essa hierarquia de herança

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- **Obs. 1:** Não faz sentido ter um objeto `Shape` no sistema, já que não sabemos calcular a área e perímetro de tal objeto.

- **Obs. 2:** Não precisamos ter um objeto `Shape` no sistema. Estamos usando essa classe **apenas para o polimorfismo**.
  - Em alguns sistemas, como é neste caso, usamos uma classe com apenas esse intuito: ganhar polimorfismo para criar métodos mais genéricos, que se encaixem a diversos objetos.

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

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- 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 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 tipos que somente contêm descrições de atributos e métodos que devem ser implementados, mas sem efetivamente implementar os métodos:
  - Classes abstratas
  - Interfaces

Classes abstratas

Métodos abstratos e classes abstratas

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

Exemplo: `abstract double area();`

Métodos abstratos e classes abstratas

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

Exemplo: `abstract double area();`

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

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

Uma classe abstrata pode ou não ter métodos abstratos. Se ela tiver, automaticamente é abstrata e deve ser declarada com o modificador `abstract`.

Métodos abstratos e classes abstratas

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

Exemplo: `abstract double area();`

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

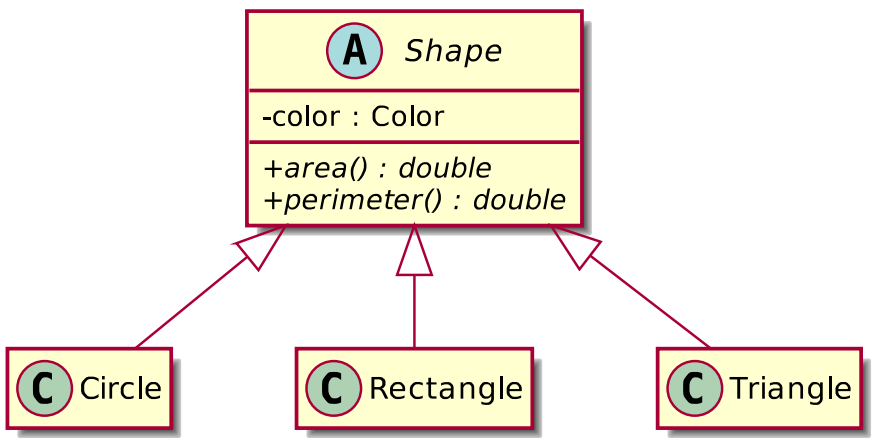
Uma classe abstrata é declarada com o modificador `abstract`.

Uma classe abstrata pode ou não ter métodos abstratos. Se ela tiver, automaticamente é abstrata e deve ser declarada com o modificador `abstract`.

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

Classe abstrata – Exemplo


```
1  public abstract class Shape {
2      private Color color;
3      public abstract double area();
4      public abstract double perimeter();
5
6      public Shape(Color color) { this.color = color; }
7      public Color getColor() { return color; }
8      public void setColor(Color color) { this.color = color; }
9
10     @Override public final String toString() {
11         return getClass().getName();
12     }
13 }
```



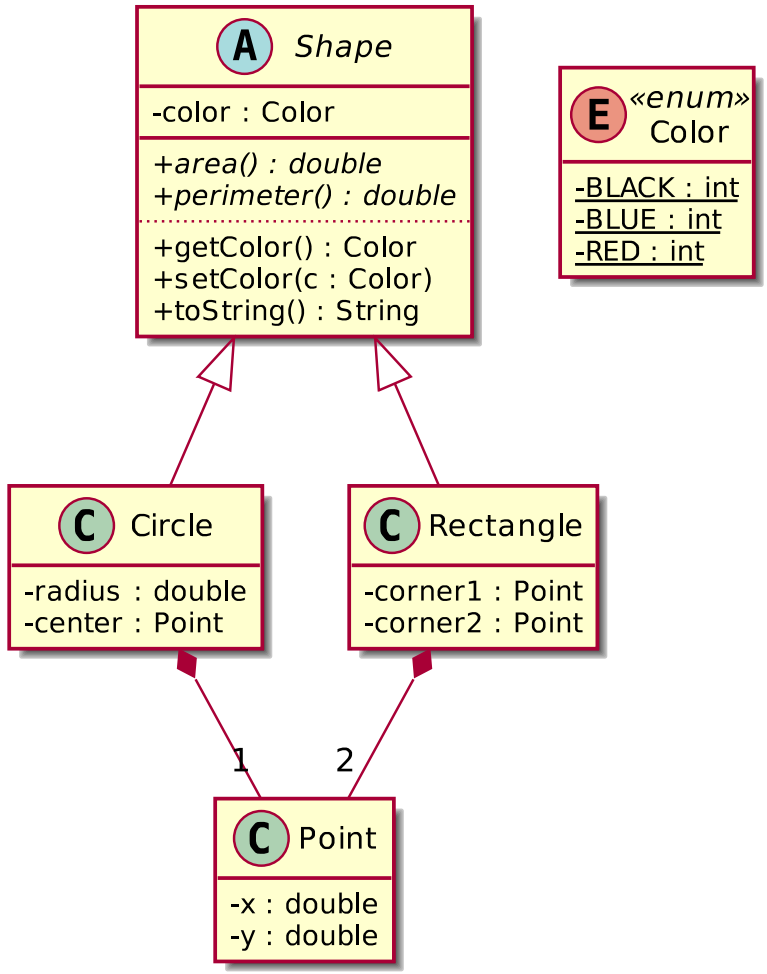
- Notação UML: *itálico*

Exercício

Fazer um programa para ler os dados de N figuras (N fornecido pelo usuário), e depois mostrar as áreas destas figuras na mesma ordem em que foram digitadas.

Enter the number of shapes: 2
Shape #1 data:
Rectangle or Circle (r/c) ? r
Color (BLACK/BLUE/RED): RED
X1 e Y1: 0.0 0.0
X2 e Y2: 5.0 6.0
Shape #1 data:
Rectangle or Circle (r/c) ? c
Color (BLACK/BLUE/RED): BLACK
Radius: 3.0
X e Y: 5.0 6.0

SHAPE AREAS:
30.00
28.27



Classes Abstratas – Observações

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

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

Classes Abstratas – Observações

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

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

- 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 – Observações

- 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.

Classes Abstratas – Observações

- 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.

Interfaces

Interface

Uma interface é um tipo que define um conjunto de operações que uma classe deve implementar.

A interface estabelece um **contrato** que a classe deve cumprir.

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

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

Interface

Uma interface é um tipo que define um conjunto de operações que uma classe deve implementar.

A interface estabelece um **contrato** que a classe deve cumprir.

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

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

```
interface Shape {
    double area();
    double perimeter();
}
```

Interface

Uma interface é um tipo que define um conjunto de operações que uma classe deve implementar.

A interface estabelece um **contrato** que a classe deve cumprir.

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

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

```
interface Shape {
    double area();
    double perimeter();
}
```

Pra quê interfaces?

- Para criar sistemas com **baixo acomplamento** e **flexíveis**.

Interface

- Uma interface não pode ser instanciada.
- Todos os métodos em uma interface são *implicitamente* `abstract` e `public`.
- Todos os atributos em uma interface são *implicitamente* `static` e `final`, devendo, portanto, ser inicializados na sua declaração.

```
interface Shape {
    double area();
    double perimeter();
}
```

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

- **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.

Interface vs. Classe Abstrata

- **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 ainda são bem diferentes de classes abstratas. Interfaces nao possuem recursos tais como construtores e atributos que não sejam constantes.

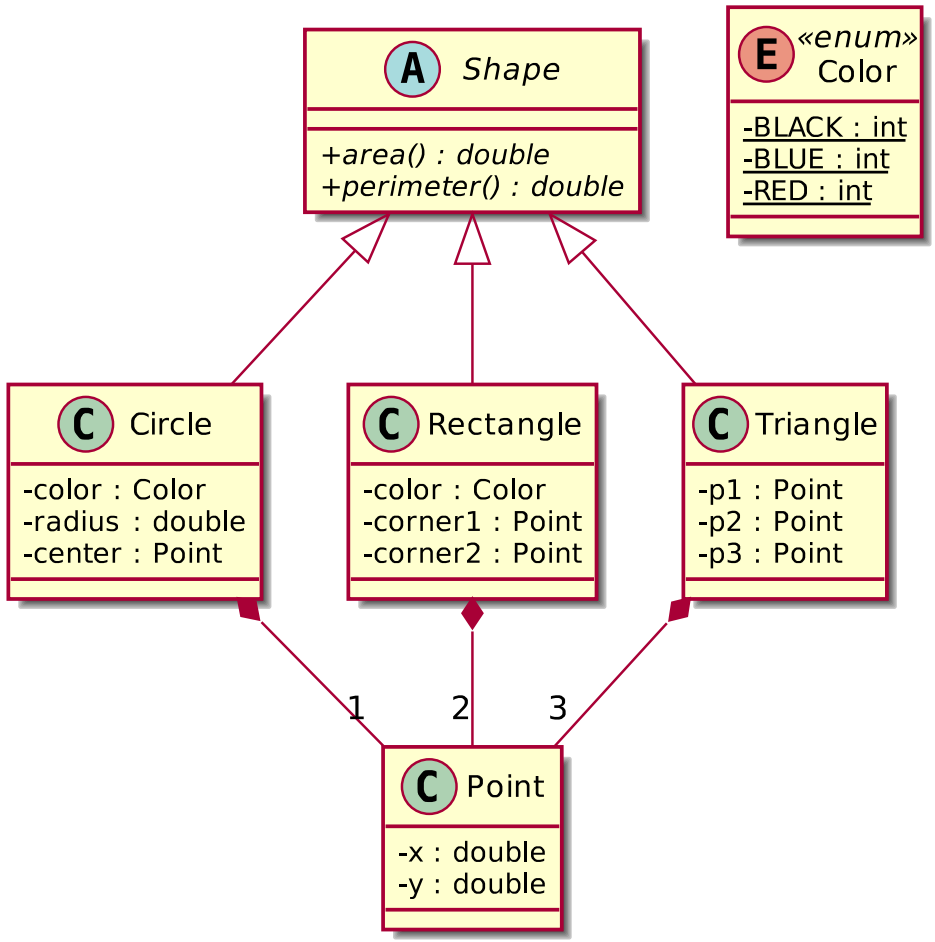
Exemplo

- Agora, queremos guardar uma lista de formas geométricas dos tipos retângulo, círculo e triângulo. Sendo que, retângulos e círculos possuem cor, mas triângulos não possuem cor.

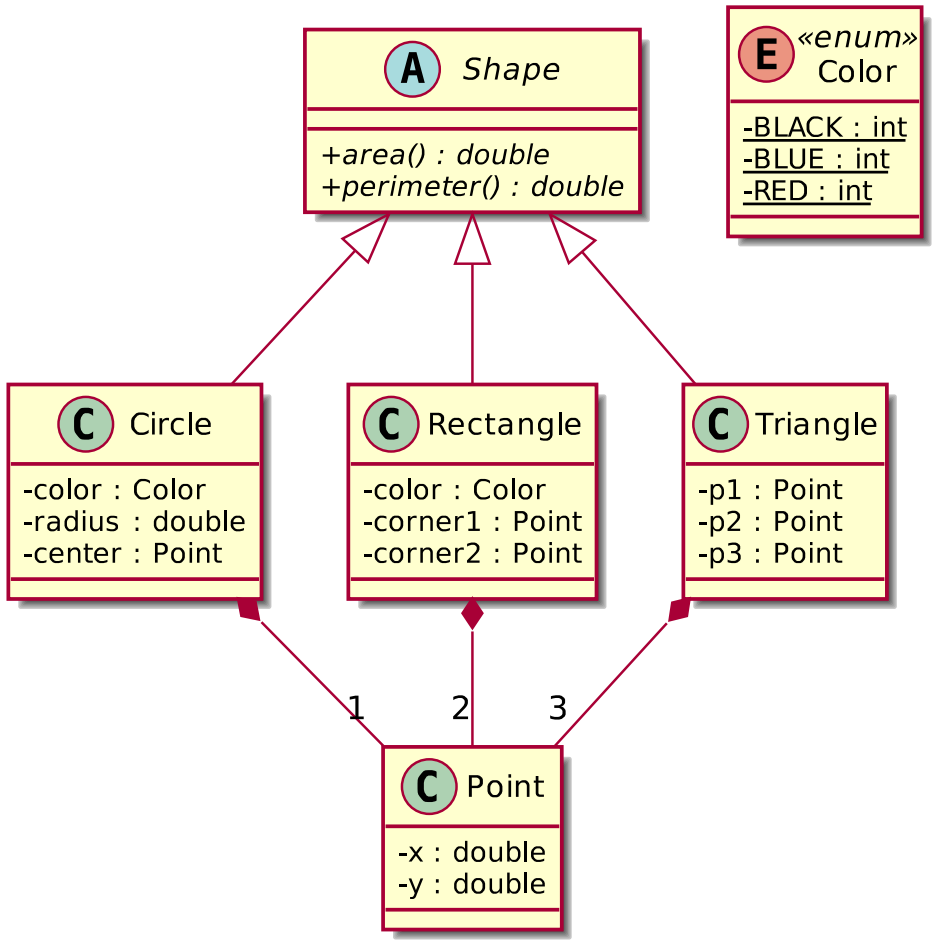
- Queremos guardar essas formas em uma mesma lista e imprimir a área de cada uma delas.

- Como modelar as classes a fim de obter esse resultado?

Exemplo — Tentativa 1

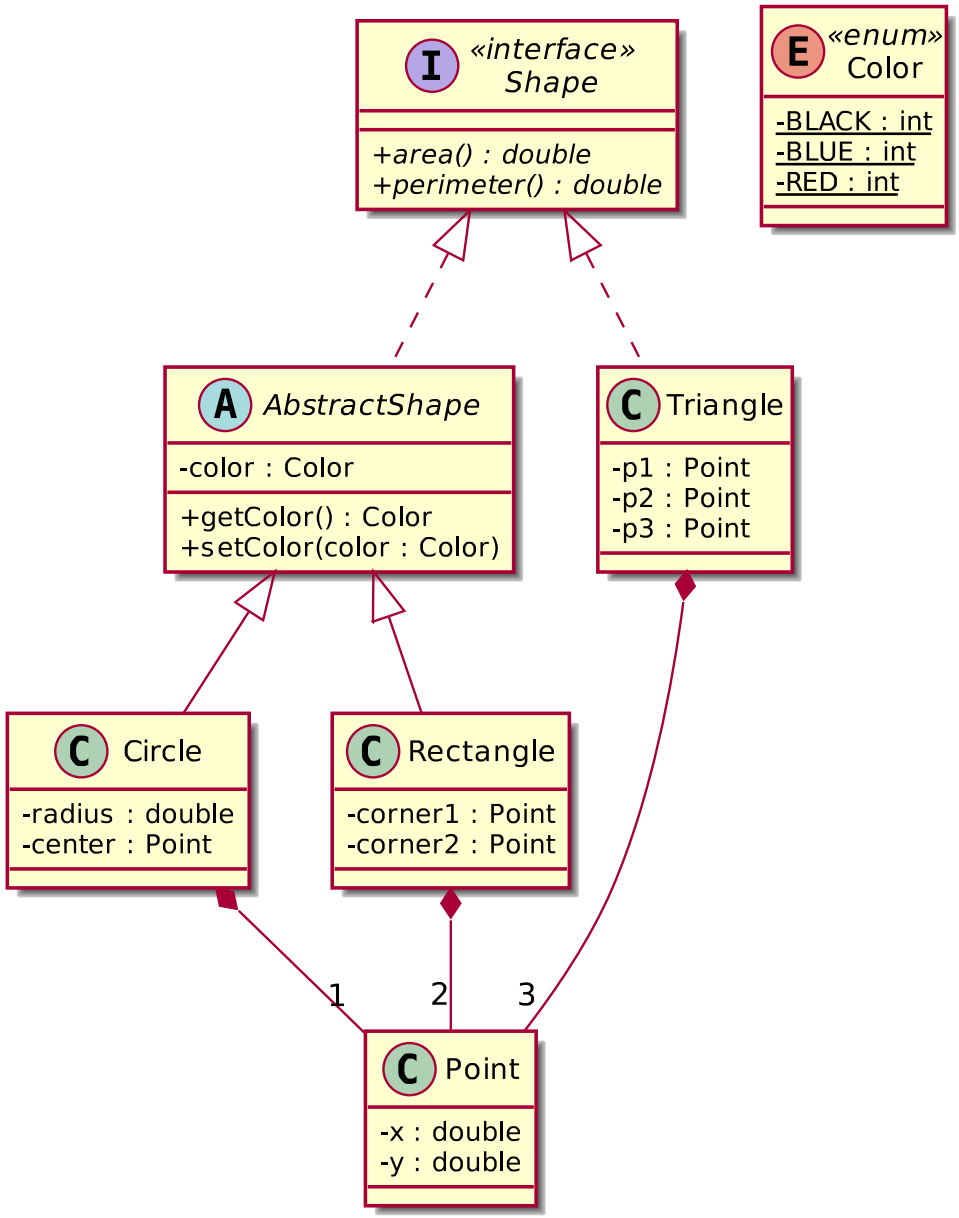


Exemplo — Tentativa 1



- Ponto negativo: o atributo `color` está repetido em `Circle` e `Rectangle`.

Exemplo — Tentativa 2



Exemplo — Tentativa 2

Fórmulas para a classe Triangle

- Fórmula da distância entre dois pontos $P = (x_1, y_1)$ e $Q = (x_2, y_2)$:

$$dist(P, Q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- Perímetro do triângulo com lados a, b, c:

$$P = a + b + c$$

- Área do triângulo com lados a, b, c (Fórmula de Heron):

$$A = \sqrt{s(s-a)(s-b)(s-c)} \qquad \text{onde} \qquad s = \frac{a+b+c}{2}$$

Interface e métodos default

Interfaces — métodos `default`

- A partir do Java SE 8 as interfaces podem conter métodos concretos.
- Se o método tiver uma implementação, antes da sua definição deve ser colocada a palavra-chave `default`.

Interfaces — métodos `default`

- A partir do Java SE 8 as interfaces podem conter métodos concretos.
- Se o método tiver uma implementação, antes da sua definição deve ser colocada a palavra-chave `default`.
- A intenção básica é prover implementação padrão para métodos, de modo a evitar:
  1) repetição de implementação em toda classe que implementa a interface
  2) a necessidade de se criar classes abstratas para prover reuso de implementação

Métodos default — Problema Exemplo

- Fazer um programa para ler uma quantia e a duração em meses de um empréstimo. Informar o valor a ser pago depois de decorrido o prazo de empréstimo.

- O valor a ser pago deve ser calculado conforme regras de juros do Brasil e também conforme regras de juros dos Estados Unidos.

- A regra de cálculo do Brasil é juro composto padrão de 2% ao ano e a regra de cálculo dos EUA é juro composto padrão de 1% ao ano.

Veja o Exemplo

Exemplo

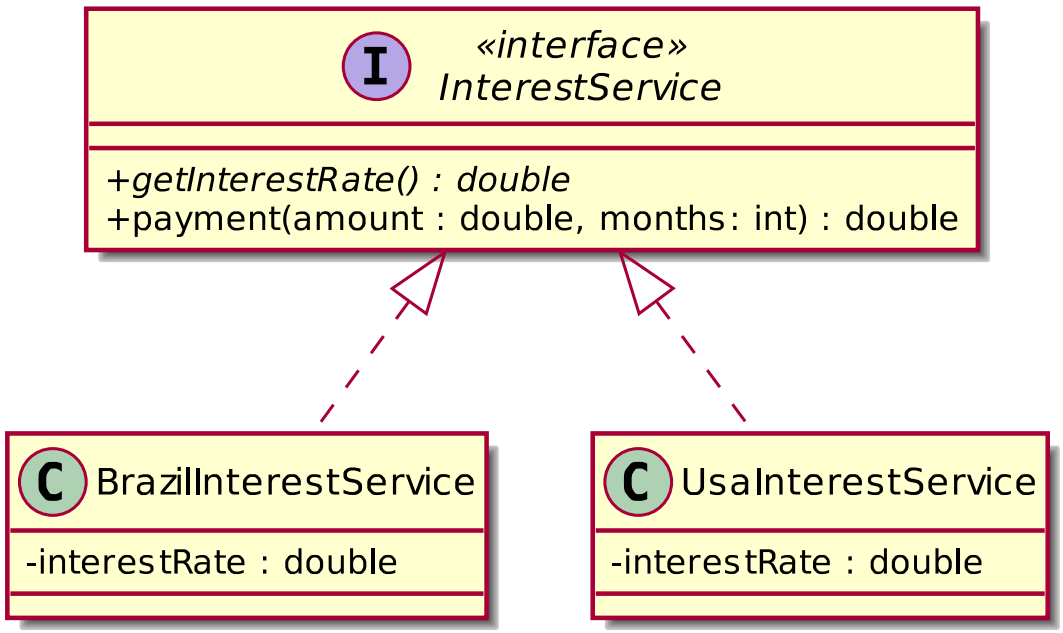

```
Amount: 200.00
Months: 3
Payment after 3 months (Brazil): 212.24
Payment after 3 months (USA): 206.06
```

Exemplo

Amount: 200.00
Months: 3
Payment after 3 months (Brazil): 212.24
Payment after 3 months (USA): 206.06

- Payment $= 200 * 1.02 * 1.02 * 1.02 = 200 * 1.02^3 = 212.2416$
- Payment $= 200 * 1.01 * 1.01 * 1.01 = 200 * 1.01^3 = 206.0602$
- Payment $=$ amount $* (1 +$ interestRate$/100)^N$

Exemplo


«interface»
InterestService
+getInterestRate() : double
+payment(amount : double, months: int) : double
BrazilInterestService
-interestRate : double
UsaInterestService
-interestRate : double

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:

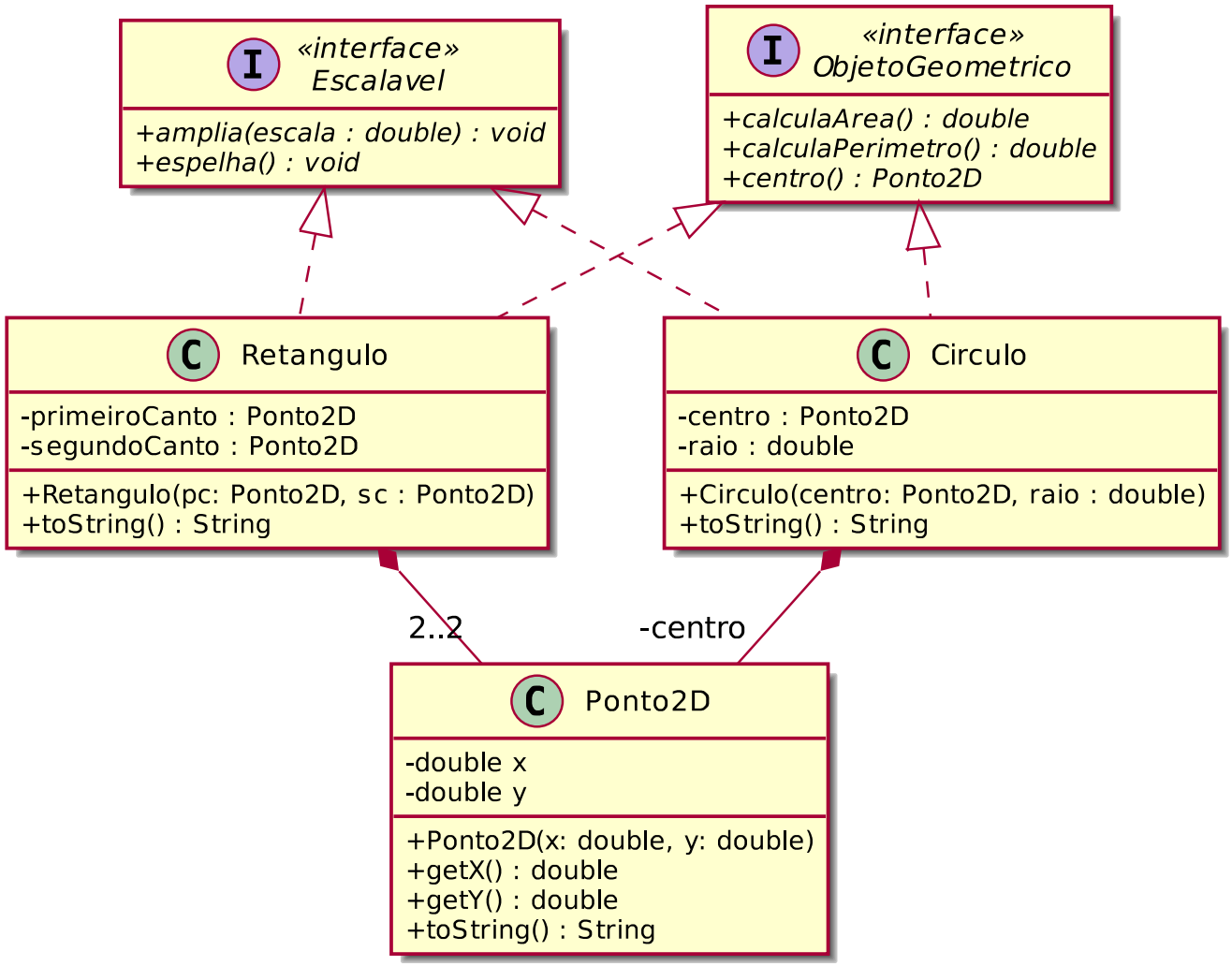
Herança múltipla usando Interfaces

- 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:** Queremos programar objetos geométricos que 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



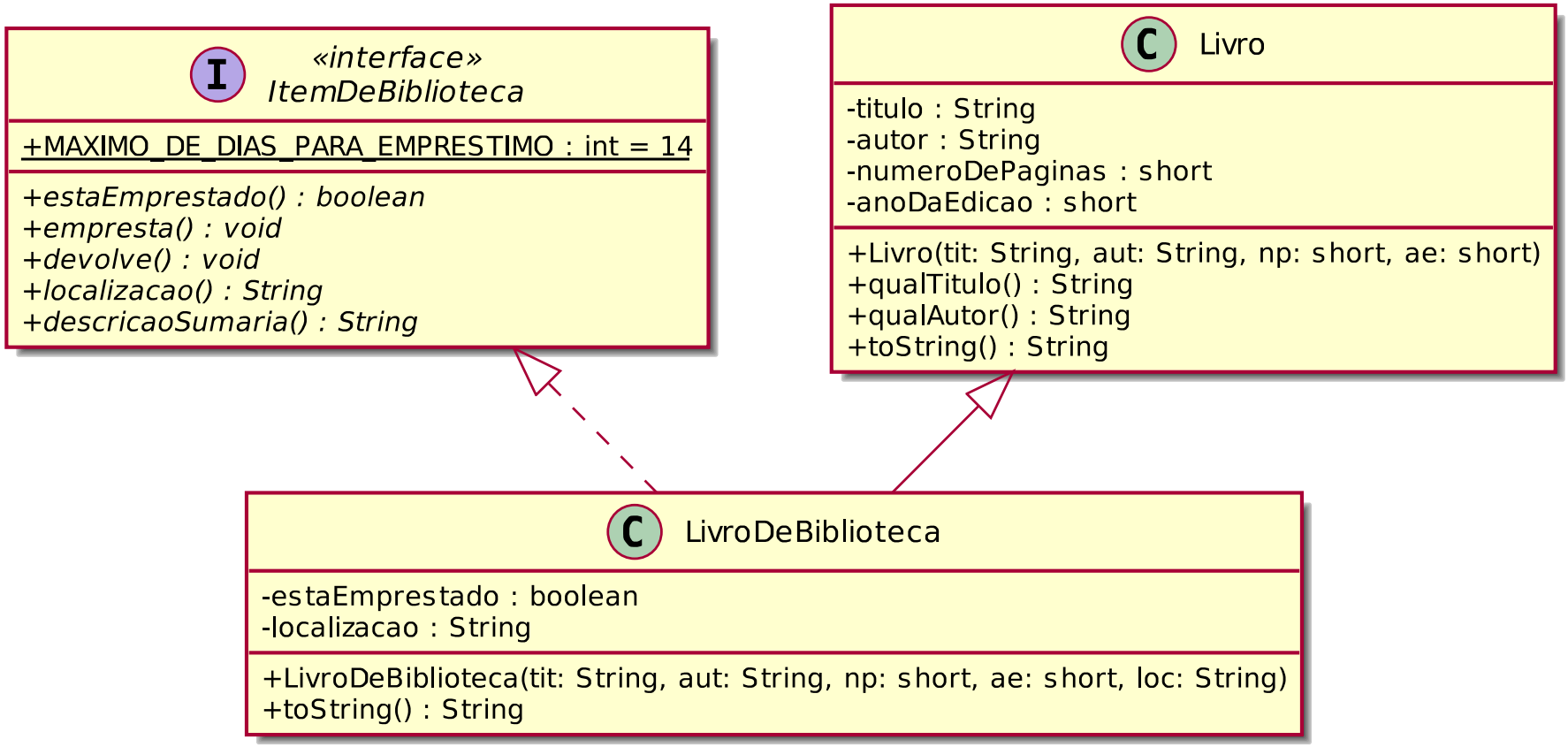
Analisar o Projeto `ObjetosEscalaveis`

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.

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.

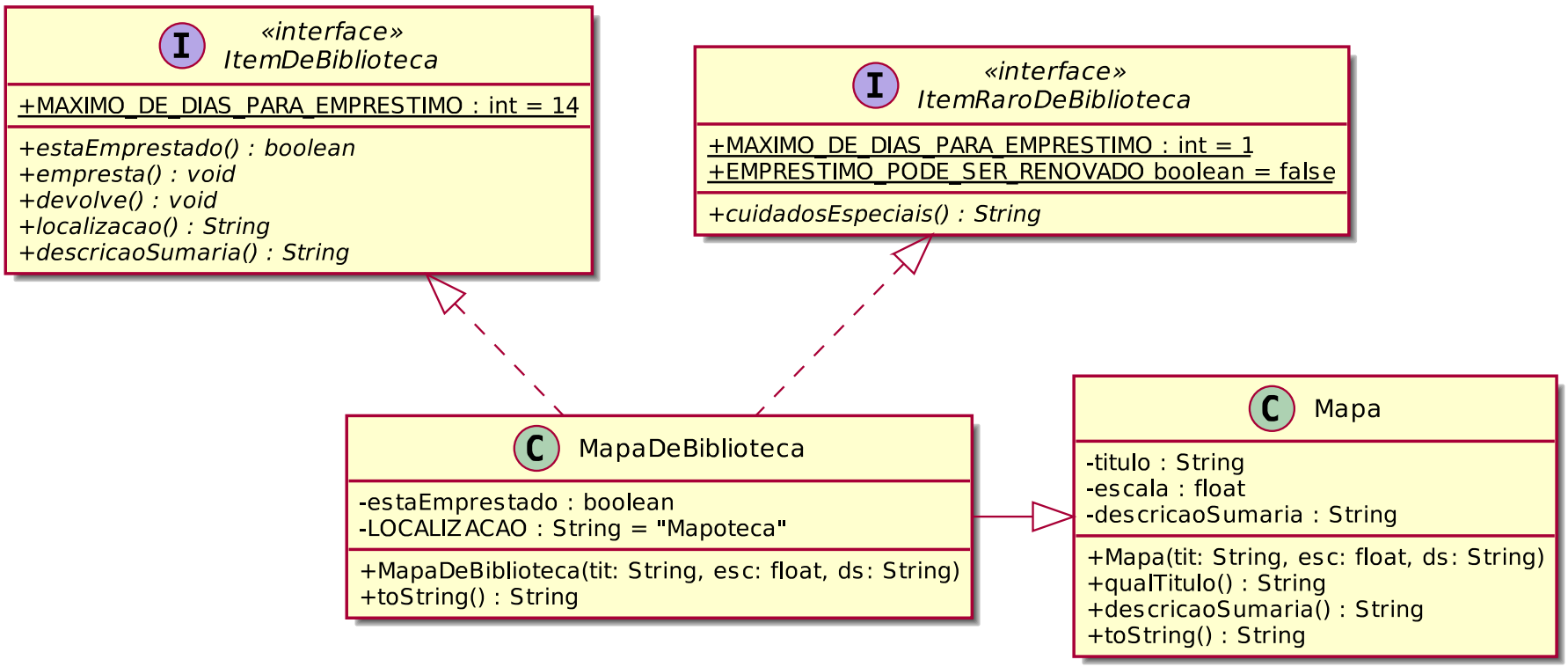
Conflitos de Herança Múltipla

- **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.

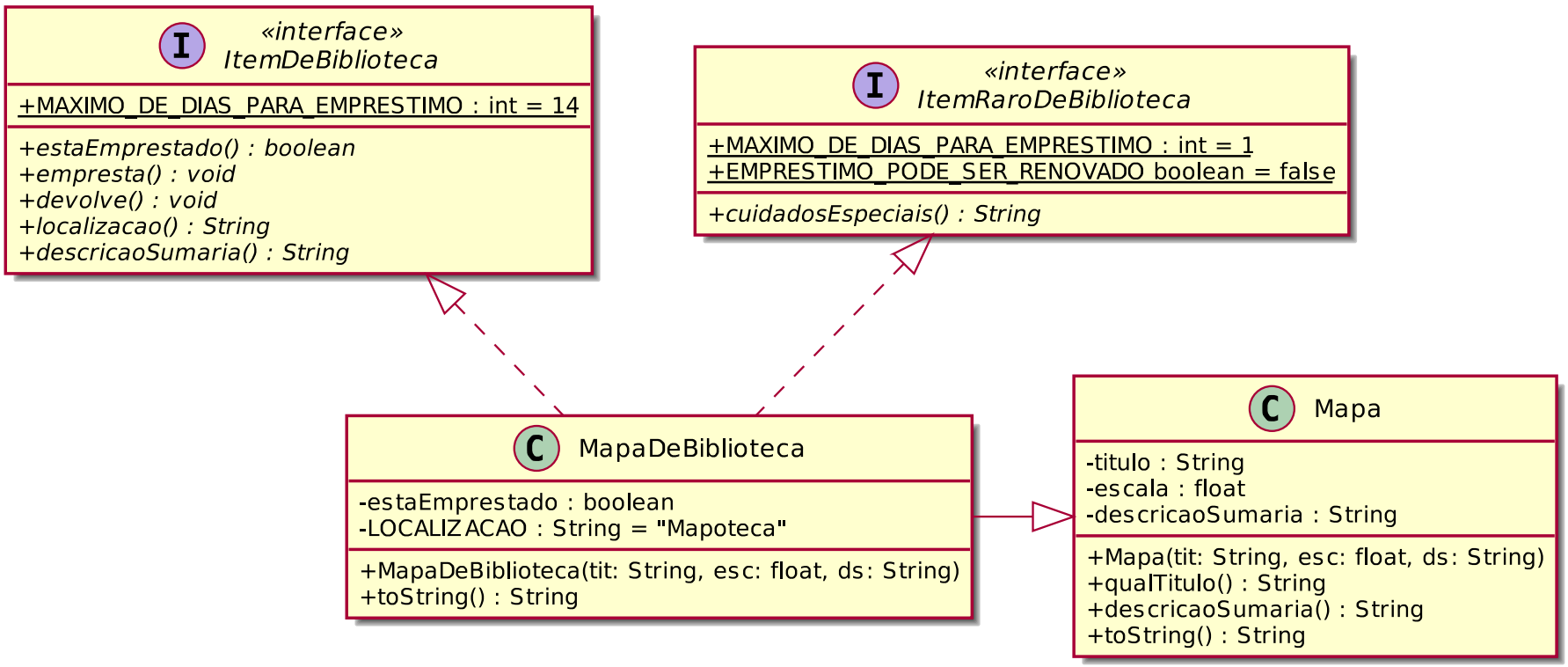
- Para definir métodos em classes que encapsulam itens raros de biblioteca, criamos uma nova interface `ItemRaroDeBiblioteca`. Também criamos as classes `Mapa` e `MapaDeBiblioteca`. Geralmente, mapas de bibliotecas são itens raros.

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`

- 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

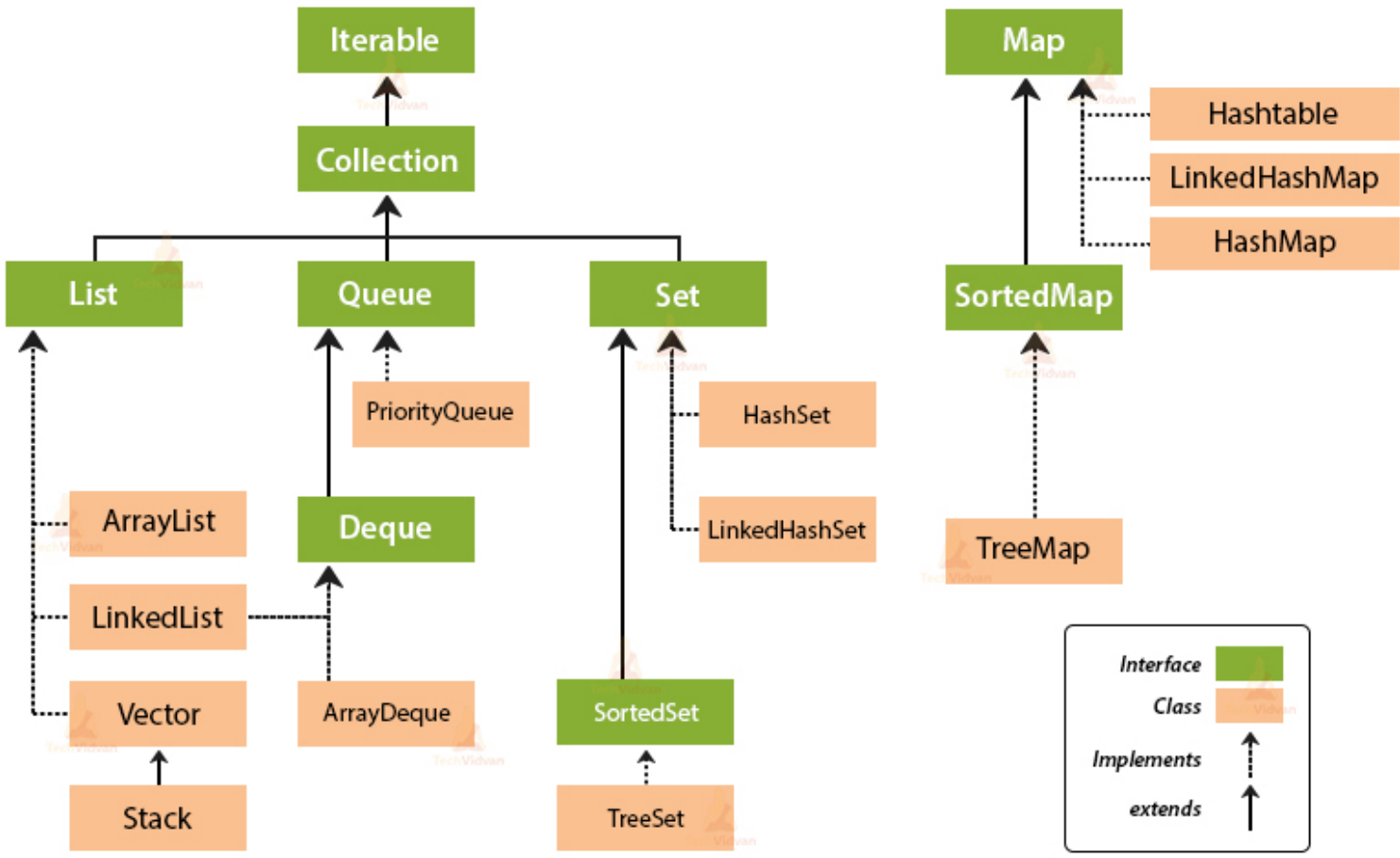
- Como já vimos, a classe **java.util.Arrays** contém vários métodos que permitem a manipulação e o processamento de arrays.

- 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 `Pessoa` 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`.

Exemplo

- Analisar os arquivos `Pessoa.java` e `App1.java` do Projeto SortingArrays



- `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.

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 arquivo `App2.java` do Projeto SortingArrays

Interface Comparator

Ordenando array de Strings

- Vimos que podemos ordenar um array de Strings usando o método estático `sort()` da classe `Arrays`, porque a classe `String` implementa a interface `Comparable`.
  - Essa ordenação compara as strings de acordo com a ordem alfabética delas.

Ordenando array de Strings

- Vimos que podemos ordenar um array de Strings usando o método estático `sort()` da classe `Arrays`, porque a classe `String` implementa a interface `Comparable`.
  - Essa ordenação compara as strings de acordo com a ordem alfabética delas.

- Suponha, agora, que queremos ordenar as strings em ordem crescente de tamanho. Como fazer isso?
  - Não podemos fazer com que a classe String implemente o método `compareTo` de duas formas diferentes.
  - Na verdade, nem sequer temos como modificar a implementação desta classe!

Ordenando array de Strings

- Vimos que podemos ordenar um array de Strings usando o método estático `sort()` da classe `Arrays`, porque a classe `String` implementa a interface `Comparable`.
  - Essa ordenação compara as strings de acordo com a ordem alfabética delas.

- Suponha, agora, que queremos ordenar as strings em ordem crescente de tamanho. Como fazer isso?
  - Não podemos fazer com que a classe String implemente o método `compareTo` de duas formas diferentes.
  - Na verdade, nem sequer temos como modificar a implementação desta classe!

- A fim de lidar com essa situação, existe uma versão sobrecarregada do método `Arrays.sort()`, cujos parâmetros são um array e um comparator (uma instância de uma classe que implementa a interface `Comparator` do Java).

A interface Comparator


```
public interface Comparator<T>
{
    int compare(T first, T second);
}
```

- A interface `Comparator` do Java, possui apenas o método `compare`. Esse método recebe dois objetos do mesmo tipo e devolve:
  - zero, se eles forem iguais;
  - um inteiro negativo, se o primeiro for menor que o segundo;
  - um inteiro positivo se o primeiro for maior que o segundo.

Comparando strings por tamanho

- Para comparar duas string por tamanho, defina uma classe que implemente a interface `Comparator<String>`.

```
class LengthComparator implements Comparator<String>
{
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

Comparando strings por tamanho

- Para comparar duas string por tamanho, defina uma classe que implemente a interface `Comparator<String>`.

```
class LengthComparator implements Comparator<String>
{
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

- Para ordenar um array de strings pelo tamanho delas, passe um objeto da classe `LengthComparator` para o método `Arrays.sort`:

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends, new LengthComparator());
```

Exemplo

- Analisar os arquivos `NomeComparator.java` e `App3.java` do Projeto SortingArrays

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM