



C125 – Programação Orientada a Objetos com Java

Interface

Prof. Phyllipe Lima
phyllipe@inatel.br

1




Agenda



- ☕ Conhecer Interfaces
- ☕ Entender a diferença entre **herdar** e **implementar**
- ☕ Exercícios

2

 Vamos resgatar nosso jogo Dark Souls

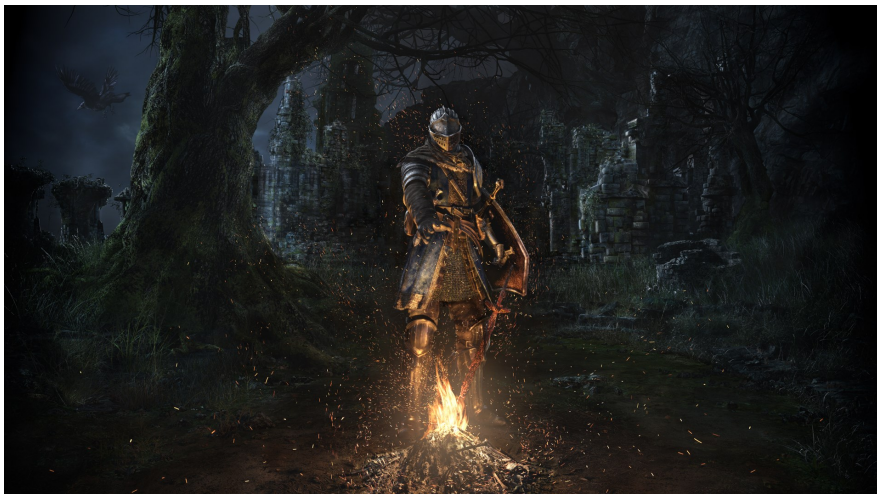


C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

3

3

Dark Souls



C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

4

4



Dark Souls – Classe Abstrata



- ☕ Criamos uma superclasse **abstrata** chamada Inimigo e outra três subclasses, herdando dela.
 - ☕ ZumbiLerdo
 - ☕ CavaleiroNegro
 - ☕ CavaleiroPrata
- ☕ Fizemos a classe Inimigo **abstrata** pois não fazia sentido uma **instância** Inimigo. Porém fazia todo sentido utilizar o tipo Inimigo como referência, permitindo assim o polimorfismo!

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

5

5



Dark Souls – Classe Abstrata



- ☕ Depois, como um exercício a parte, fizemos uma classe **abstrata** Jogador para representar
 - ☕ Mago
 - ☕ Guerreiro
 - ☕ Arqueiro
- ☕ Agora iremos conectar esses dois pedaços, e adicionalmente, iremos criar um novo tipo de jogador, Sacerdote!
- ☕ Faremos a classe Jogador **abstrata**. O método atacar() não será abstrato e continuará recebendo uma referência para Inimigo.

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

6

6



Classe Jogador!



```
public abstract class Jogador {  
  
    protected String nome;  
    protected double vida;  
    protected String tipoArma;  
  
    public Jogador(String nome, double vida, String tipoArma) {  
        this.nome = nome;  
        this.vida = vida;  
        this.tipoArma = tipoArma;  
    }  
  
    public void atacar(Inimigo inimigo){  
        inimigo.tomarDano();  
        System.out.println("Jogador atacou o inimigo " + inimigo.getNome());  
    }  
}
```

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

7

7



Jogadores que Curam!



- ☞ Conforme o projeto do jogo avança, novos requisitos surgem!
- ☞ O projetista do jogo decidiu que alguns tipos de jogadores (isto é, classes que herdam de Jogador) podem recuperar vida.
- ☞ Ficou definido que apenas Mago e Sacerdote podem fazer isso.
- ☞ Como poderíamos implementar esse requisito?

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

8

8

Jogadores que Curam!



- ☕ Pode parecer adequado inserir um método chamado recuperarVida() na classe Jogador.
- ☕ Mas lembre-se que todas as classes que herdam de Jogador também terão esse comportamento recuperarVida().
- ☕ E os requisitos deixam bem claro que apenas Mago e Sacerdote podem ter esse comportamento

```
public abstract class Jogador{

    protected String nome;
    protected double vida;
    protected String tipoArma;

    public Jogador(String nome, double vida, String tipoArma) {
        this.nome = nome;
        this.vida = vida;
        this.tipoArma = tipoArma;
    }

    //Não é boa ideia, pois Guerreiro e Arqueiro
    //Também conseguirão utilizar esse método
    public void recuperarVida() {
        this.vida += 50;
    }
}
```

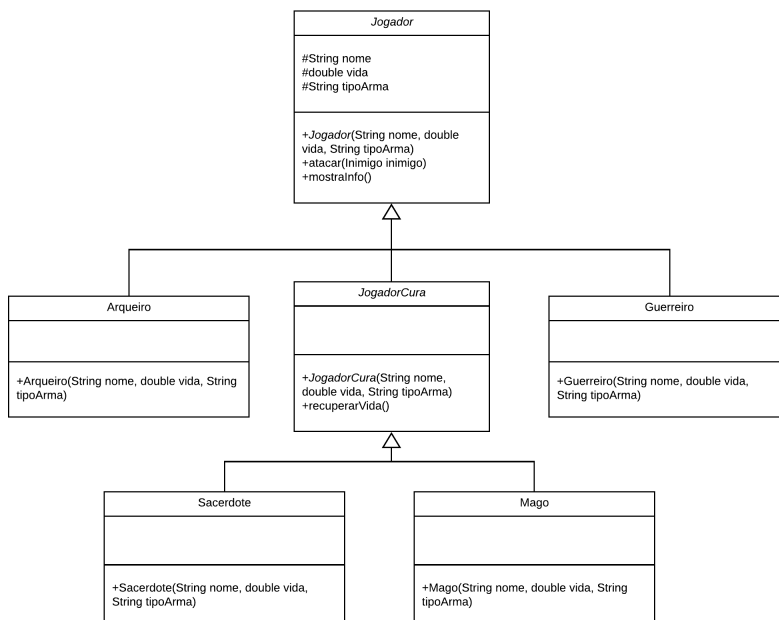


Jogadores que Curam!



- Outra ideia seria criar uma camada intermediária de herança
- Poderíamos criar uma nova classe chamada JogadorCura contendo o método recuperarVida() e apenas as classes Mago e Sacerdote **herdam** de JogadorCura. E JogadorCura, por sua vez, **herda** de Jogador.
- Assim, Mago e Sacerdote podem ser referenciados como JogadorCura ou Jogador.
- Em termos UML teríamos algo como mostrado no próximo slide (aproximadamente)

11



12



Jogadores que Curam!



- ☕ A princípio esse código continuaria funcionando de forma adequada apenas enquanto criamos novos jogadores.
- ☕ Exemplo, imagine que teremos agora um novo tipo chamado Bruxo, e ele também é capaz de recuperar sua vida. Ele pode herdar de JogadorCura e está tudo resolvido. O projeto continua coeso afinal, o Bruxo também **é um** jogador.
- ☕ O problema irá surgir se quisermos que algo que **não seja** um jogador possa recuperar sua vida também.



Inimigos também Curam!



- ☕ Uma ideia que pode surgir é “Vamos fazer CavaleiroDePrata **herdar** de JogadorCura, uma vez que no geral Inimigos e Jogadores estão parecidos”
- ☕ Péssima ideia. Primeiramente por questões de organização e modelagem, podemos dizer que CavaleiroDePrata **é um** Jogador? Não! Absolutamente não. Isso não faz sentido e deixaria nosso programa totalmente confuso.
- ☕ Mesmo que no código isso **poderia** funcionar, nosso programa iria ficar muito prejudicado em termos de evolução. Novos requisitos surgem a todo momento, e se fizermos essa modelagem misturando classes que representam “coisas” diferentes, podemos criar um código impossível (ou muito difícil) de evoluir e dar manutenção.



Inimigos Não São Jogadores!



- ☞ Observe que nossa classe Jogador possui um método atacar() que recebe um Inimigo. Faria sentido o CavaleiroDePrata herdar isso?
- ☞ Assim é importante utilizar a herança apenas quando existe a relação “**é um**”. Caso contrário estamos ignorando a orientação a objeto e fazendo nossas classes serem apenas depósito de código.
- ☞ Mas ainda não resolvemos o problema de termos Inimigos e Jogadores que podem recuperar a vida.
- ☞ O que precisamos é criar um contrato **do que** algumas classe podem fazer. E toda classe, que desejar, poderia **implementar** esse contrato.

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

15

15



Criando Um Contrato



- ☞ Perceba que escrevi **o que fazer** e não **como fazer**.
- ☞ Todo método possui duas características:
 - ☞ O que ele faz, isto é, sua assinatura: modificador, tipo de retorno e parâmetros
 - ☞ Como ele faz, isto, o código implementado no seu corpo
- ☞ Saber o que uma classe **faz** nos permite mais generalização, pois é isso que deixamos exposto. **Como ela faz** é algo que fica encapsulado dentro do corpo dos métodos.

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

16

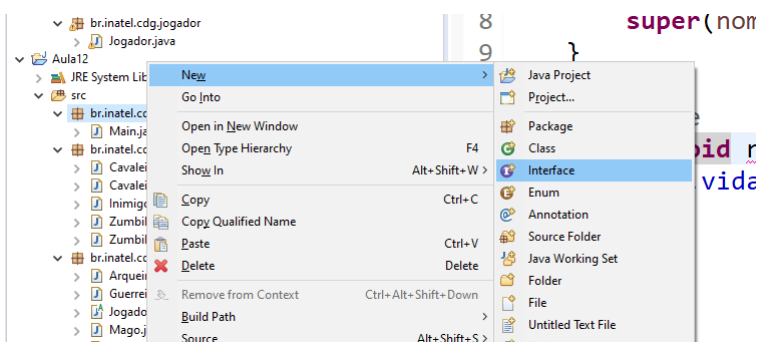
16

Criando Uma Interface

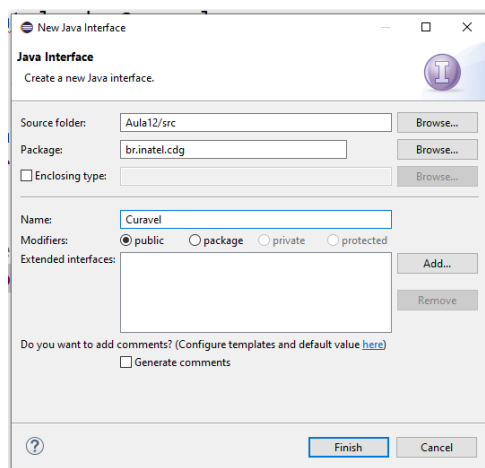


- Em muitas linguagens orientada a objetos podemos criar uma **interface** que define **o que** uma classe deve fazer, **mas não como fazer**.
- Vamos criar uma interface chamada Curavel. E todo Jogador ou Inimigo que deseja também **ser um** Curavel deve **implementar** essa interface e **obrigatoriamente** dizer **como irá fazer**.
- A interface Curavel só diz **o que fazer**. Cada classe que a **implementar** irá decidir **como fazer**.

Criando Interface no Eclipse



Criando Interface no Eclipse



C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

19

19

Interface Curavel

```
package br.inatel.cdg;

public interface Curavel {

    public void recuperarVida();

}
```

- ☕ Perceba que uma interface é um **tipo** de dado.
- ☕ Todos os seus métodos são **públicos** e **abstratos** por padrão
- ☕ Não possuem nenhuma implementação, isto é, dizem apenas **o que fazer** e não **como fazer**.
- ☕ Desde o Java 8 temos como criar métodos **default** em interfaces, que permite implementações (**como fazer**). Em outro momento veremos isso.

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

20

20

Interface Curavel



☕ Com a nossa interface pronta, podemos fazer qualquer outra classe **implementá-la** através da palavra chave **implements**

☕ Considere o exemplo para o Mago

```
public class Mago extends Jogador implements Curavel{

    public Mago(String nome, double vida, String tipoArma) {
        super(nome, vida, tipoArma);
    }
}
```

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

21

21

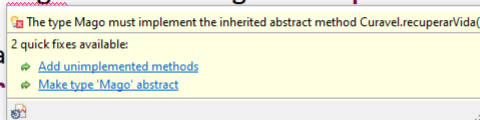
Interface Curavel



☕ Perceba que não compila. Veja a mensagem de erro fornecida pelo Eclipse

```
public class Mago extends Jogador implements Curavel{

    public Mago(String nome, double vida, String tipoArma) {
        super(nome, vida, tipoArma);
    }
}
```



☕ As classes que implementam Curavel, **obrigatoriamente**, precisam implementar todos os métodos. Afinal, por padrão eles são abstratos!

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

22

22



Interface Curavel



```
public class Mago extends Jogador implements Curavel{

    public Mago(String nome, double vida, String tipoArma) {
        super(nome, vida, tipoArma);
    }

    @Override
    public void recuperarVida() {
        this.vida += 10;
    }
}
```

☞ Como exemplo da classe Mago, o método recupera 10 pontos de vida

☞ Cada classe poderá implementar de acordo com suas especificações



```
public class CavaleiroPrata extends Inimigo implements Curavel {

    public CavaleiroPrata(String nome, double vida, String tipoArma) {
        super(nome, vida, tipoArma);
    }

    public void ataqueForte() {
        System.out.println("Ataque Forte!");
    }

    @Override
    public void recuperarVida() {
        this.vida += 40;
    }
}
```



☞ CavaleiroPrata também é Curavel!

☞ Ele **é um** Inimigo e **é um** Curavel!

☞ Então pode ser referenciado como Curavel e Inimigo?



```
public static void main(String[] args) {

    ZumbiLerdo zumbi = new ZumbiLerdo("Zumbi Lerdo", 50, "Espada Curta");
    CavaleiroNegro cavNegro =
        new CavaleiroNegro("Cavaleiro Negro", 150, "Espada Longa");
    CavaleiroPrata cavPrata =
        new CavaleiroPrata("Cavaleiro Prata", 175, "Silver Sword");

    //Não compila, pois ZumbiLerdo não é um Curavel
    Curavel inimigoCuravel = zumbi;
    //Compila! Pois CavaleiroPrata É UM Curavel
    Curavel inimigoCuravel2 = cavPrata;
    //E também É UM Inimigo. Ou seja, pode ser referenciado como ambos!
    //E claro, como um CavaleiroPrata
    Inimigo inimigo = cavPrata;
}
```

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

25

25



Interfaces

- Classes que **implementam** Curavel também podem **ser tratadas** como Curavel.
- Podem existir métodos do nosso programa que recebem variáveis do tipo Curavel, assim como temos métodos que recebem variáveis do tipo Inimigo e foi possível passar ZumbiLerdo, CavaleiroNegro e CavaleiroPrata
- Em outras palavras, como interfaces podemos utilizar o polimorfismo também
- Métodos que recebem variáveis do tipo Curavel podem receber instâncias de Mago, Sacerdote e CavaleiroPrata (de acordo com nosso projeto!)

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

26

26



Interfaces vs Classe Abstrata



☞ É natural surgir a pergunta “Mas qual diferença entre Classe Abstrata e Interface?”.

- ☞ Ambas não podem ser instanciadas
- ☞ Ambas definem novos tipos
- ☞ Classes podem implementar mais de uma interface (***implements***), mas podem ***herdar*** apenas de uma superclasse (***extends***)
- ☞ Uma classe abstrata que possui apenas métodos abstratos públicos e nenhum membro, na prática se tornou uma interface
- ☞ Como uma interface possui apenas métodos, a mensagem que queremos passar é que estamos criando um novo tipo que só possui comportamento. Por exemplo o Curavel.
- ☞ A Classe Abstrata podemos ter estado também, como Inimigo e Jogador.

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

27

27



Interfaces vs Classe Abstrata



- ☞ Interface: Estamos abstraindo comportamento
 - ☞ Curavel é uma ação que corresponde a recuperar a vida
- ☞ Classe Abstrata: Estamos abstraindo estado e comportamento
 - ☞ Jogador: Possui estado (nome, vida) e comportamento (sabe atacar)

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

28

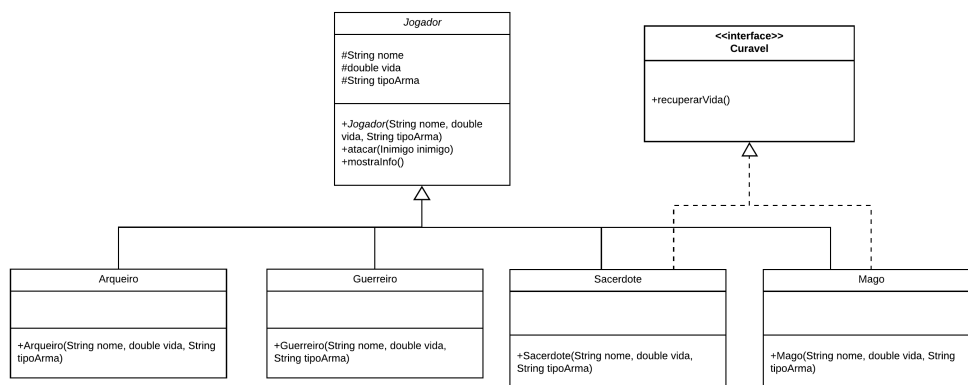
28

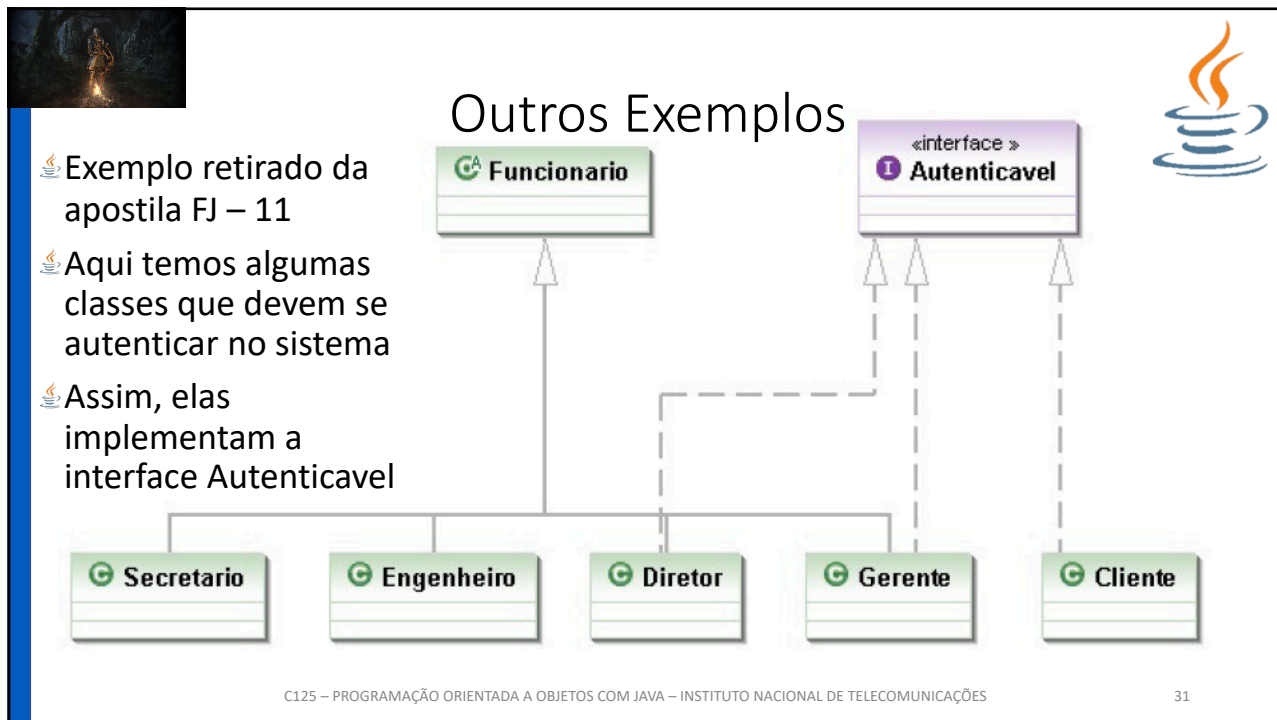
UML - Interface



- No diagrama UML, para representar uma interface, usamos a palavra <<interface>> e logo abaixo colocamos o nome, por exemplo, Curavel.
- Para indicar implementação usamos a seta branca, mas a linha tracejada. A ponta da seta fica na interface, semelhante a herança
- Observe no diagrama UML que a interface não possui área para membros, reforçando que ela possui apenas comportamento.

UML - Interface





31

Exercício



- Termine de implementar o exemplo apresentado nessa aula
- Considere que Mago, Sacerdote e CavaleiroPrata irão implementar Curavel.
- Faça testes com um método main

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

32

Material Complementar



 Capítulo 11 da apostila FJ-11
 Interfaces

C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

33

33

Resolução dos Exercícios



<https://github.com/phillima-inatel/C125>



C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

34

34



C125 – Programação Orientada a Objetos com
Java

Interface

Prof. Phyllipe Lima
phyllipe@inatel.br

