

Inatel



C125 – Programação Orientada a Objetos com
Java

Herança e Polimorfismo

Prof. Phyllipe Lima
phyllipe@inatel.br

1

Inatel



C125 – Programação Orientada a Objetos com
Java

Herança e Polimorfismo



Prof. Phyllipe Lima
phyllipe@inatel.br

2

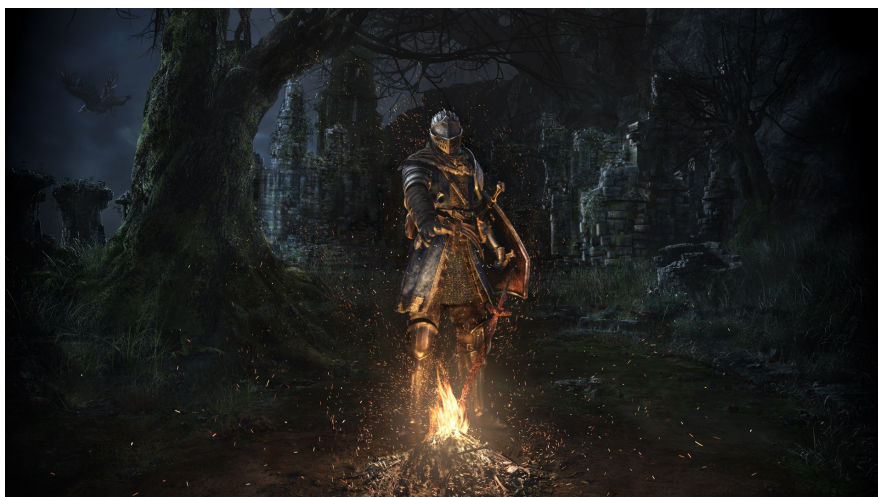


Agenda



- ☕ Entender o conceito de Herança
- ☕ Entender como aplicar o Polimorfismo
- ☕ Fazer sobrescrita de métodos
- ☕ Reutilizar código

Dark Souls





Modelando os Inimigos de Dark Souls



☞ Suponha que o jogo Dark Souls tenha 3 tipos de Inimigos

- ☞ Zumbi Lerdo
- ☞ Cavaleiro Negro
- ☞ Cavaleiro de Prata

☞ Vamos escrever a classe para modelar o Zumbi Lerdo



```
package br.inatel.cdg.inimigo;
```

```
public class ZumbiLerdo {
```

```
    private String nome;  
    private double vida;  
    private String tipoArma;
```

```
    //Construtor
```

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


```
    public void atacando() {  
        System.out.println("Atacando o jogador!");  
    }
```

```
    public void tomarDano() {  
        System.out.println("Tomando dano");  
    }
```



Modelando os Inimigos de Dark Souls



 Vamos agora escrever a classe para modelar o Cavaleiro Negro

7



```
package br.inatel.cdg.inimigo;

public class CavaleiroNegro {

    private String nome;
    private double vida;
    private String tipoArma;

    //Construtor
    public CavaleiroNegro(String nome, double vida, String tipoArma) {
        this.nome = nome;
        this.vida = vida;
        this.tipoArma = tipoArma;
    }

    public void atacando() {
        System.out.println("Atacando o jogador!");
    }

    public void ataqueRapido() {
        System.out.println("Atacando rapidamente!");
    }

    public void tomarDano() {
        System.out.println("Tomando dano");
    }
}
```



8



Modelando os Inimigos de Dark Souls



- ☕ Elas estão bem parecidas correto?
- ☕ Se olhar rapidamente, parecem a mesma classe!
- ☕ Porém ela possui o método *atacarRapido()* que o ZumbiLerdo não possui
- ☕ Precisamos repetir todo esse código para cada novo inimigo que queremos modelar no nosso jogo?



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

9

9



Modelando os Inimigos de Dark Souls



- ☕ Podemos ver que essas classes compartilham muitas características!
- ☕ Deve existir algum recurso para escrevermos menos código
- ☕ E se essas classes compartilhassem uma mesma abstração, algo como uma classe "Inimigo"? E em seguida pudessem **herdar** os membros e métodos?



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

10

10



Herança!

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

11

11




Herança

- ☕ É um recurso do paradigma orientado a objetos
- ☕ Permite que classes possam **herdar** métodos e membros de uma classe Mãe, também conhecida como superclasse. As classes que herdam são classes filhas, ou subclasse.
- ☕ Como ficaria nosso exemplo do Dark Souls?
- ☕ Vamos primeiro criar uma classe Inimigo com os membros e métodos comuns.

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

12

12



```
package br.inatel.cdg.inimigo;

public class Inimigo {


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

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


    public void atacando() {
        System.out.println("Atacando o jogador!");
    }

    public void tomarDano() {
        System.out.println("Tomando dano");
    }


}
```



13



Modelando o Inimigo



- ☕ Observe um novo modificador chamado **protected**. Ele possui uma visibilidade mais limitada que o **public** e menos restrita que **private**. Com esse modificador, somente a própria classe e as subclasses podem ter acesso a esses membros.
- ☕ Normalmente esse modificador é utilizado nos membros das superclasses.

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

14

14

Modelando o Inimigo



- ☞ Observe que na classe `Inimigo`, não existe método `ataqueRapido()`, pois é específico do `CavaleiroNegro`. Na classe `inimigo` deixamos apenas o que for comum a **TODOS** os inimigos!
- ☞ Para que as classes `ZumbiLerdo` e `CavaleiroNegro` possam **herdar** da classe `Inimigo`, usamos a palavra chave **`extends`** (para a linguagem Java).
- ☞ Outras linguagens OO como C# e C++ possuem outra sintaxe para que as classes possam **herdar** das superclasses.
- ☞ Vamos ao resultado!

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

15

15

```
package br.inatel.cdg.inimigo;

public class ZumbiLerdo extends Inimigo {

    //Construtor
    public ZumbiLerdo(String nome, double vida, String tipoArma) {
        super(nome, vida, tipoArma);
    }
}

package br.inatel.cdg.inimigo;

public class CavaleiroNegro extends Inimigo {

    //Construtor
    public CavaleiroNegro(String nome, double vida, String tipoArma) {
        super(nome, vida, tipoArma);
    }

    public void ataqueRapido() {
        System.out.println("Atacando rapidamente!");
    }
}
```



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

16

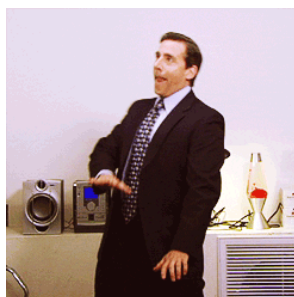
16



Modelando o Inimigo com Herança



- ☕ As classes ZumbiLerdo e CavaleiroNegro ficaram bem menores não é mesmo?
- ☕ Agora fica bem mais fácil evoluir esse software



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

17

17



Modelando o Inimigo com Herança



- ☕ Observações
 - ☕ A palavra chave **super** se refere a superclasse, nesse exemplo a Inimigo. Ou seja, estamos chamando o construtor da superclasse Inimigo.
 - ☕ Na classe CavaleiroNegro, colocamos o método **ataqueRapido()**, pois é uma **especialização** dessa classe. Não existe razão para colocarmos ela na classe ZumbiLerdo.
- ☕ Vamos testar essas classes!

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

18

18

Modelando o Inimigo com Herança



```
public static void main(String[] args) {
    ZumbiLerdo zumbiLerdo =
        new ZumbiLerdo("Lerdao", 50, "Espada Curti");
    CavaleiroNegro cavaleiro =
        new CavaleiroNegro("Cavaleiro", 100, "Espada Longa");

    zumbiLerdo.atacando();

    cavaleiro.atacando();
    cavaleiro.ataqueRapido();
}
```

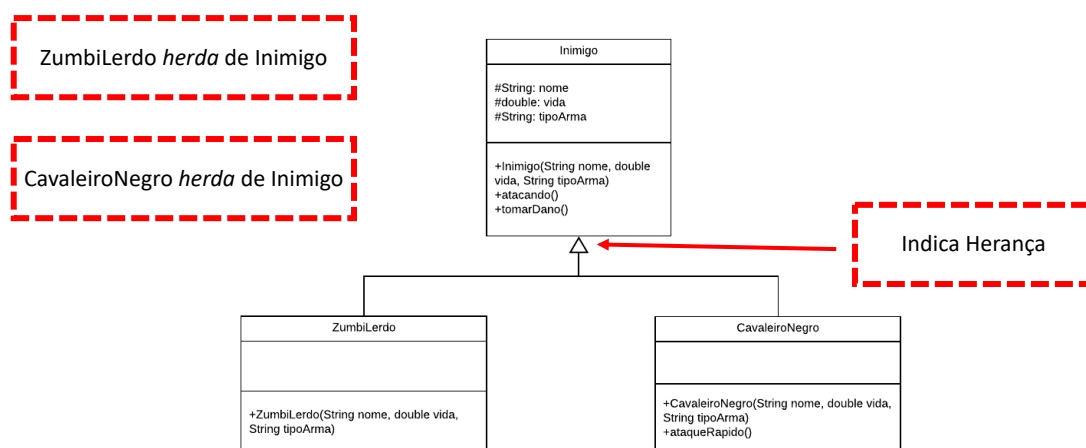
Atacando o jogador!
Atacando o jogador!
Atacando rapidamente!

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

19

19

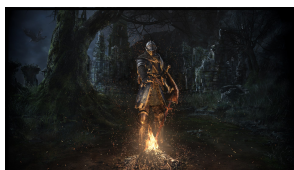
UML com Herança



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

20

20

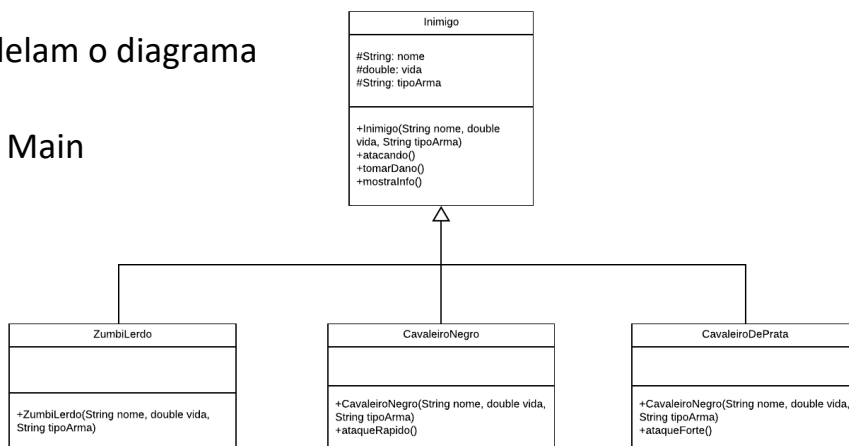


Exercício 1 – Dark Souls



☕ Crie classes que modelam o diagrama UML ao lado

☕ Faça testes na classe Main



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

21

21

Sobrescrita de Métodos



☕ Observe que as mensagens exibidas no método **atacando()** foram as mesmas para ambas as instâncias (ZumbiLerdo e CavaleiroNegro). Temos como especializar esse comportamento?

☕ Sim! Através da sobrescrita!

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

22

22

Sobrescrita de Métodos



```
package br.inatel.cdg.inimigo;

public class ZumbiLerdo extends Inimigo {

    //Construtor
    public ZumbiLerdo(String nome, double vida, String tipoArma) {
        super(nome, vida, tipoArma);
    }

    @Override
    public void atacando() {
        System.out.println("Zumbi Lerdo Atacando!");
    }
}
```

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

23

23

Sobrescrita de Métodos



- ☞ Observe que colocamos uma **anotação** `@Override` em cima do método “atacando()” para indicar que estamos sobrescrevendo um método da superclasse. Fizemos isso na classe ZumbiLerdo
- ☞ Podemos assim especializar comportamento nas subclasses
- ☞ Vamos testar essa chamada

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

24

24

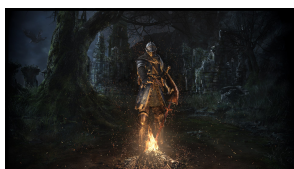


```
public static void main(String[] args) {
    ZumbiLerdo zumbiLerdo =
        new ZumbiLerdo("Lerdao", 50, "Espada Curti");
    CavaleiroNegro cavaleiro =
        new CavaleiroNegro("Cavaleiro", 100, "Espada Longa");

    zumbiLerdo.atacando();
    cavaleiro.atacando();
    cavaleiro.ataqueRapido();
}
```



Zumbi Lerdo Atacando!
Atacando o jogador!
Atacando rapidamente!



Exercício 2 – Dark Souls



☞ Faça uma modificação no código do Exercício 1, e permita que cada classe que herde de Inimigo, faça a sobrescrita do método *atacando()*, e personalize a mensagem

Polimorfismo



- ☞ Quando dizemos que ZumbiLerdo **herda** da classe Inimigo, dizemos um ZumbiLerdo **É UM** Inimigo.
- ☞ Com essa definição, se uma classe Jogador é capaz de causar dano a um Inimigo, ele pode causar dano em qualquer classe que **herda** de Inimigo.
- ☞ Se toda subclasse de Inimigo **É UM** Inimigo, então podemos salvar uma referência de ZumbiLerdo em uma variável do tipo Inimigo?
- ☞ Sim!
- ☞ Veja o código a seguir

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

27

27

Polimorfismo



```
public static void main(String[] args) {
    Inimigo zumbiLerdo =
        new ZumbiLerdo("Lerdao", 50, "Espada Curti");
    Inimigo cavaleiro =
        new CavaleiroNegro("Cavaleiro", 100, "Espada Longa");

    zumbiLerdo.atacando();
    cavaleiro.atacando();
}
```

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

28

28

Polimorfismo



- ☕ Perceba que agora as instâncias de CavaleiroNegro e ZumbiLerdo foram salvas em uma variável do tipo Inimigo
- ☕ Mas isso não significa que essas instâncias mudaram de tipo. O que mudou foi a forma como elas estão sendo referenciadas. Elas continuam sendo instâncias de CavaleiroNegro e ZumbiLerdo, mas suas referências podem ser armazenadas em qualquer variável que seja de uma superclasse, como a Inimigo por exemplo!

Polimorfismo



```
public static void main(String[] args) {
    Inimigo zumbiLerdo =
        new ZumbiLerdo("Lerdao", 50, "Espada Curti");
    Inimigo cavaleiro =
        new CavaleiroNegro("Cavaleiro", 100, "Espada Longa");

    zumbiLerdo.atacando();
    cavaleiro.atacando();
}
```

- ☕ O que será impresso ao executar o código? O método atacando() original na classe Inimigo, ou o método atacando() que foi sobrescrito nas classes ZumbiLerdo e CavaleiroNegro?

Polimorfismo



```
public static void main(String[] args) {
    Inimigo zumbiLerdo =
        new ZumbiLerdo("Lerdao", 50, "Espada Curti");
    Inimigo cavaleiro =
        new CavaleiroNegro("Cavaleiro", 100, "Espada Longa");

    zumbiLerdo.atacando();
    cavaleiro.atacando();
}
```

Zumbi Lerdo Atacando!
Cavaleiro Negro Atacando!

☕ O que será impresso ao executar o código? O método `atacando()` original na classe `Inimigo`, ou o método `atacando()` que foi sobrescrito nas classes `ZumbiLerdo` e `CavaleiroNegro`?

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

31

31

Polimorfismo



- ☕ Dado que as instâncias são de fato `ZumbiLerdo` e `CavaleiroNegro`, o método chamado será o presente nessas classes e não ao da classe `Inimigo`.
- ☕ Se não tivéssemos feito uma sobrescrita, seria chamado o método original definido na classe `Inimigo`.
- ☕ Porém, percebe-se que como as instâncias estão salvas em variáveis do tipo `Inimigo`, não podemos invocar métodos específicos das instâncias.
- ☕ Exemplo: Não conseguimos invocar o método `ataqueRapido()`, definido na classe `CavaleiroNegro`, pois a classe `Inimigo` não “conhece” esse método

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

32

32

Polimorfismo



```
public static void main(String[] args) {
    Inimigo zumbiLerdo =
        new ZumbiLerdo("Lerdao", 50, "Espada Curti");
    Inimigo cavaleiro =
        new CavaleiroNegro("Cavaleiro", 100, "Espada Longa");

    zumbiLerdo.atacando();
    cavaleiro.atacando();

    cavaleiro.ataqueRapido();//Não compila
```

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

33

33

Polimorfismo



- ☕ Mas como fazemos para invocar o método ataqueRapido()?
- ☕ Primeiro precisamos verificar se uma instância é de um tipo, e depois trocar o tipo de variável de referência.
- ☕ Podemos usar o operador *instanceof()*. Com esse operador podemos testar se uma instância é de um determinado tipo. Se for, podemos fazer o *casting* e em seguida trabalhar com métodos específicos.

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

34

34



Polimorfismo

```
public static void main(String[] args) {
    Inimigo zumbiLerdo =
        new ZumbiLerdo("Lerdao", 50, "Espada Curti");
    Inimigo cavaleiro =
        new CavaleiroNegro("Cavaleiro", 100, "Espada Longa");

    zumbiLerdo.atacando();
    cavaleiro.atacando();

    if(cavaleiro instanceof CavaleiroNegro) {
        //Salva cavaleiro em uma variável do tipo CavaleiroNegro
        CavaleiroNegro cav = (CavaleiroNegro)cavaleiro;
        cav.ataqueRapido();//Agora compila!
    }
}
```

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

35

35



Polimorfismo

- ☕ Qual a vantagem do Polimorfismo?
- ☕ Com ele podemos criar classes que lidam apenas com as abstrações (ou superclasse) e assim podemos deixar nossas funcionalidades mais genéricas.
- ☕ Considere uma classe Jogador, que possui um método que recebe um parâmetro do tipo Inimigo e faz alguma operação nessa variável.

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

36

36

Polimorfismo



```
public class Jogador {
    public void atacar(Inimigo inimigo) {
        System.out.println("Atacando o Inimigo: " + inimigo.getNome());
    }
}
```

- ☞ Qualquer instância que *herda* de Inimigo pode ser passada como parâmetro!
- ☞ Isso traz poder e flexibilidade para escrever programas.
- ☞ Perceba como fica fácil evoluir esse software, criando novos tipos de inimigos para o jogo Dark Souls

Polimorfismo



- ☞ Talvez você esteja pensando, qual a razão disso? Por que simplesmente não criar todas as classes como Inimigo? Nem precisaríamos de herança e polimorfismo. Dentre as várias razões, uma delas está relacionada as camadas do software.
- ☞ Exemplo: Podem existir trechos do código que faz sentido trabalhar com instâncias do tipo ZumbiLerdo, mas em outras partes do código, pode fazer sentido atuar nessa mesma instância, mas como se fosse um Inimigo!



Exercício 3 – Dark Souls PT3

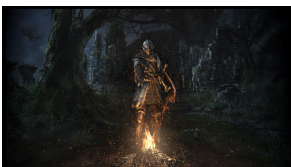


- ☕ Considere o UML do próximo Slide.
- ☕ Crie uma classe Jogador, que recebe um Inimigo e causa dano. Faça com o que o próprio inimigo invoque o método tomarDano(). Imprima também o nome do Inimigo que está perdendo vida.
- ☕ Faça o método tomarDano() diminuir a vida do Inimigo.

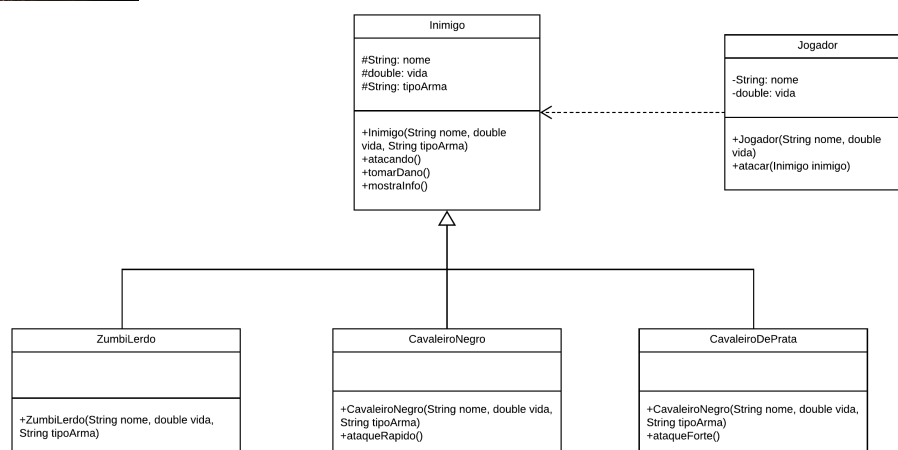
C125 – PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA – INSTITUTO NACIONAL DE TELECOMUNICAÇÕES

39

39



Exercício 3 – Dark Souls PT3



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

40

40

Resolução dos Exercícios



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




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

41

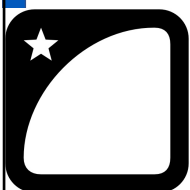
41


Material Complementar



 Vídeo aula sobre Herança

<https://youtu.be/oCHs5WXwBJI>



 Capítulo 9 da apostila FJ-11

 Herança, Reescrita e Polimorfismo

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

42

42

Inatel

C125 – Programação Orientada a Objetos com
Java

Herança e Polimorfismo

Prof. Phyllipe Lima
phyllipe@inatel.br

