Inatel

C125 – Programação Orientada a Objetos com Java

Classe Abstrata

Prof. Phyllipe Lima
phyllipe@inatel.br

1

Agenda

- Conhecer classes abstratas
- Entender porque existem e quando utilizá-las
- Utilizando métodos abstratos
- Exercícios

2

- Vamos resgatar nosso jogo Dark Souls

3

Dark Souls

4

Dark Souls

- Criamos uma superclasse chamada Inimigo e outra três subclasses, herdando dela.
  - ZumbiLerdo
  - CavaleiroNegro
  - CavaleiroPrata
- Seria possível criarmos instâncias de Inimigo?

5

```java
public class Main {

    public static void main(String[] args) {

        Inimigo inimigo = new Inimigo("Inimigo", 30, "Arma Comum");

        inimigo.atacando();

    }
}
```

- Sim, é possível! O código compila e executa sem erros
- Mas faz sentido termos instâncias de Inimigo?
- Veja que é diferente de termos ***referências*** do tipo Inimigo

6

```java
public class Main {

    public static void main(String[] args) {

        Inimigo inimigo = new Inimigo("Inimigo", 30, "Arma Comum");

        inimigo.atacando();

    }
}
```

- Sim, é possível! O código compila e executa sem erros
- Mas faz sentido termos instâncias de Inimigo?
- Veja que é diferente de termos ***referências*** do tipo Inimigo

7

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

    //Instância de Inimigo
    Inimigo inimigo = new Inimigo("Inimigo", 30, "Arma Comum");

    //Instância de ZumbiLerdo sendo referenciado como Inimigo
    Inimigo zumbi = new ZumbiLerdo("Zumbi Lerdo", 50, "Espada Curta");

}
```

- No código acima temos um exemplo de uma instância de Inimigo e de uma instância de ZumbiLerdo sendo armazenado como uma ***referência*** para Inimigo!

8

- Quando pensamos em instâncias, pensamos em objetos concretos. Que fazem sentido realizar comportamento!
- Quando falamos nos inimigos, nós imaginamos um Zumbi Lerdo, um Cavaleiro Negro e assim por diante.
- Mas apenas um "Inimigo" parece algo ***abstrato*** para termos uma instância desse tipo!
- Mas por que então criamos a classe Inimigo?

- Em primeiro lugar, para evitar repetir código!
- Criamos classes como ZumbiLerdo e CavaleiroNegro, todas elas ***herdando*** de Inimigo e reusando sua estrutura (membros e métodos). Com isso, economizamos bastante código. Observe essas classes

```java
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!");
    }
}
```

```java
public class CavaleiroNegro extends Inimigo {

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

    @Override
    public void atacando() {
        System.out.println("Cavaleiro Negro Atacando!");
    }
    public void ataqueRapido() {
        System.out.println("Atacando rapidamente!");
    }
}
```

- E podemos também criar novos tipos de inimigos, como CavaleiroPrata, EsqueloFogo e etc. Herdando da classe ***Inimigo***. Isso favorece e ***evolução*** do nosso software.
- E com herança, podemos utilizar o poder do polimorfismo. Criamos métodos genéricos que sabem apenas lidar com a superclasse e os método nela presente.
- Reveja a classe Jogador

```
public class Jogador{

    private String nome;
    private double vida;


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

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

- Observe que o método ***atacar(Inimigo inimigo)*** recebe instâncias ***referenciadas*** ou ***do tipo*** Inimigo. Ele não precisa conhecer nenhuma classe que ***herda*** de Inimigo. Isso também favorece a ***evolução*** do software

Classes Abstratas

- Resgatando a questão. Faz sentido ter instâncias do tipo Inimigo? Não
- Mas faz ***todo*** sentido termos ***referências*** do tipo Inimigo. Afinal, é assim que o método ***atacar(Inimigo inimigo)*** funciona. Ele recebe ***referências*** para Inimigo
- Concluímos que criamos a classe Inimigo apenas para ser ***referências*** (variáveis) e não ***instâncias*** (objetos na memória)***.***
- Para isso, podemos dizer que ela uma classe ***abstrata***
- No Java e C# temos a palavra chave ***abstract*** para esse fim.
- Observe a nova classe ***abstrata*** Inimigo

13

```java
public abstract 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!");
    }
```


14

Classes Abstratas

- Quando fazemos uma classe ***abstract***, estamos passando a seguinte informação.
  - Não desejamos instanciar essa classe
  - Ela deve ser uma superclasse e suas subclasses serão instanciadas
  - Ele deve ser usada como referência para permitir o polimorfismo
- O compilador Java garante que ela não será instanciada. Mas pode ser referenciada normalmente.
- Apenas suas subclasses poderão ser instanciadas

Classes Abstratas

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

    //Instância de Inimigo. NÃO COMPILA
    Inimigo inimigo = new Inimigo("Inimigo", 30, "Arma Comum");

    //Instância de ZumbiLerdo sendo referenciado como Inimigo
    //Compila!
    Inimigo zumbi = new ZumbiLerdo("Zumbi Lerdo", 50, "Espada Curta");

}
```

Classes Abstratas

- Seria correto dizer que toda superclasse deve ser abstrata?
- Não!!!!!
- Depende do escopo do seu projeto e de suas abstrações. Não é obrigatório fazer toda superclasse abstrata
- Poderíamos fazer ZumbiLerdo ser uma superclasse também. E criarmos novos tipos de zumbis a partir dela. Então teríamos ZumbiLerdo herdando de Inimigo e ZumbiLerdFogo (por exemplo) herdando de ZumbiLerdo. Mas, faz sentido no jogo ter uma instância de ZumbiLerdo.
- Assim, não há razão para faze-la ser abstrata

Classes Abstratas

- Repare que podemos ter várias camadas (gerações) de Herança
- Mas não podemos ter uma mesma classe herdando de ***mais de uma*** classe em uma única declaração.

```
//Compila
//Podemos fazer a seguinte analogia:
//ZumbiLerdoFogo é filho de Zumbilerdo e neto de Inimigo
public class ZumbiLerdoFogo extends ZumbiLerdo {

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

}
```

Classes Abstratas

- O Código abaixo não compila, pois estamos tentando, na mesma declaração fazer uma classe herdas de outras duas!

```
//Não Compila
public class ZumbiLerdoFogo extends ZumbiLerdo, Inimigo {

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

}
```


Classes Abstratas

- Agora que conhecemos a classe abstrata podemos pensar que então que seu uso está restrito a polimorfismo e evitar repetição de código.
- Mas podemos fazer isso com uma classe normal, tomando cuidado de manter nosso código consistente.
- Vamos pensar no método atacando() na superclasse Inimigo.

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

Método Abstrato

- Imagine que as subclasses resolvessem não implementar.
- Nesse caso, cada subclasse utilizaria o comportamento da superclasse, ou seja, imprimir “Atacando o jogador”.
- E se quiséssemos forçar que cada subclasse sobrescreva o método atacando()? Afim de garantir comportamento específico?
- Quando temos uma classe abstrata, podemos ter também um método abstrato. Isto é, não possui implementação na superclasse, e toda subclasse é obrigada a implementar.
- Vamos deixar esse método abstrato, na classe abstrata Inimigo

Método Abstrato

```
//Não compila
//Metodo abstrato não pode ter implementação
//em sua definição
public abstract void atacando() {
    System.out.println("Atacando o jogador!");
}
```

- Vamos na classe ZumbiLerdo e remover o método atacando()
- Perceba que o código não irá compilar

```
//Agora compila :)
public abstract void atacando();
```

Método Abstrato

- Vamos na classe ZumbiLerdo e remover o método atacando()
- Perceba que o código não irá compilar

```
//Não Compila
//Precisamos, obrigatoriamente, implementar atacando
//O Eclipse nos ajuda nessa tarefa
public class ZumbiLerdo extends Inimigo {

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


}
```


Método Abstrato

- Perceba a mensagem de erro gerada pelo Eclipse

```
//Não Compila
//Precisamos, obrigatoriamente, implementar atacando
//O Eclipse nos ajuda nessa tarefa
public class ZumbiLerdo extends Inimigo {

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


}
```

The type ZumbiLerdo must implement the inherited abstract method Inimigo.atacando()
2 quick fixes available:
Add unimplemented methods
Make type 'ZumbiLerdo' abstract
Press 'F2' for focus

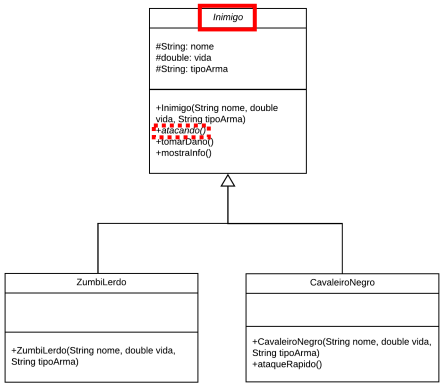
- Se clicarmos em “Add unimplemented methods”, o Eclipse já colocará o corpo desses métodos na classe ZumbiLerdo

Método Abstrato

```java
public class ZumbiLerdo extends Inimigo {

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

    @Override
    public void atacando() {
        //Agora de a sua implementação!
    }


}
```


UML

- No diagrama UML, classes e métodos abstratos aparecem com a fonte itálica



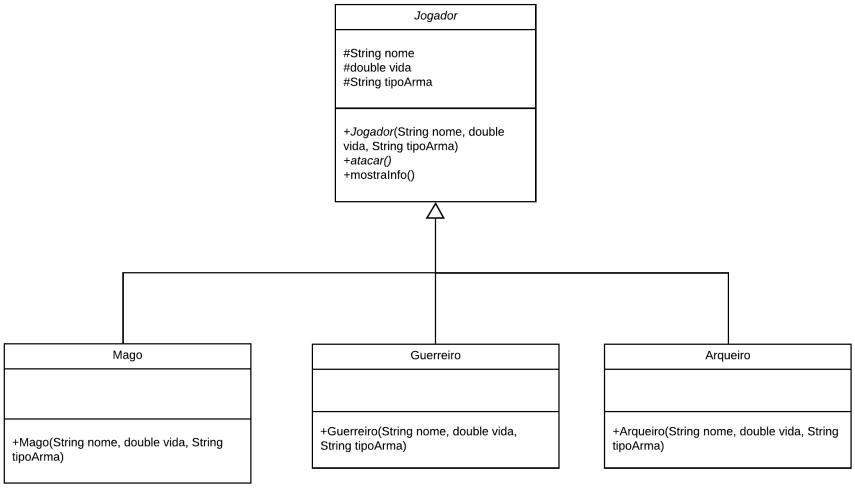
Outros Exemplos

- Seguem outros exemplos que ***pode*** fazer sentido ser uma classe abstrata
  - Pessoa
  - Funcionário
  - Mamífero
  - Veículo
  - Animal
  - ....

Exercício

- Crie classes Java para modelar o UML no próximo slide
- Crie uma classe Main, com método main() para fazer o teste!
- O Mago utiliza um cajado, o Guerreiro um machado e Arqueiro um arco.
- Cada classe que herda Jogador, deve implementar o seu método *atacar()*. Pode apenas imprimir mensagens
- Esse exercício está separado do exemplo apresentado nessa aula
- A classe Jogador e o método atacar() são abstratos

Exercício



29

Resolução dos Exercícios

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

30

Material Complementar

- Capítulo 10 da apostila FJ-11
  - Classes Abstratas

Inatel

C125 – Programação Orientada a Objetos com Java

Classe Abstrata

Prof. Phyllipe Lima
phyllipe@inatel.br