# Slides 12: Polimorfismo

Baseado nos slides oficiais do livro Java – Como programar – Deitel e Deitel – 10ª edição

# Introdução

- Conceito
  - Polimorfirmo = poli + morpho = várias formas
    - Mesmo código tem comportamento diferente dependendo do contexto
- Objetivos
  - Simplificar a programação usando objetos de classes diferentes mas que tem a mesma superclasse podem ser tratados como se fossem objetos da superclasse
    - Objetos das classes Assalariado e Horista podem ser tratados como se fossem objetos da classe Colaborador
    - Objetos das classes Circulo e Quadrado tratados como objetos da classe Figura2D
    - Objetos de Cachorro, Ave e Peixe tratados como objetos da classe Animal

# Introdução

- Benefícios
  - É possível implementar uma funcionalidade, mesmo sem saber previamente detalhes da implementação das subclasses
    - Pode-se criar um método para calcular o Imposto de Renda de um trabalhador sem saber como seu salário é calculado, desde que ele tenha um método getSalario
    - Usar um desenho qualquer em um painel desde que ele tenha o método draw
    - Emitir o som de um animal em um jogo desde que ele tenha um método emiteSom

# Funcionamento do polimorfismo

- Cria-se uma superclasse com métodos que são compartilhados com as subclasses
- Uma subclasse pode sobrescrever esses métodos e criar os seus próprios
- No código, cria-se uma referência da superclasse que é atribuída a um objeto da subclasse
- Ao chamar um método, mesmo sendo declarado como sendo da superclasse, o objeto executa o método referente ao objeto da subclasse
- Esse processo é realizado em tempo de execução e e chamado de vinculação dinâmica (dynamic binding)

# Exemplo 1

```
public class Animal {
    public String emiteSom( ) { return " ";  }
}
public class Leao extends Animal {
    public String emiteSom( ) { return "Roarrr";    }
}
public class Peixe extends Animal {
    public String emiteSom( ) { return "Blurp";     }
}
...
Animal a1 = new Leao( ) , a2 = new Peixe ( );
System.out.println(a1.emiteSom( ));
System.out.println(a2.emiteSom( ));
```

# Exemplo 2

```
public class Vitoria {
    public String toString() {  return "Vitoria "; }
}
public class VitoriaPorPontos extends Vitoria { ...
    public String toString() {  return "Vitoria por pontos (" +
pontosVencedor + " - " + pontosPerdedor + ")."; }
}
public class VitoriaPorPontos extends Vitoria{ ...
    public String toString() {  return "Vitoria por nocaute aos "
+ segundos + "s  do " + round +"º round ("+golpe+").";}
}
Vitoria v1 = new VitoriaPorPontos(50, 45), v2 = new
VitoriaPorNocaute(1, 13, "cruzado de direita");
System.out.println(v1 + "\n" + v2);
```

# Classes abstratas

- Em algumas situações é cômodo criar superclasses, mesmo que nunca sejam instanciados objetos dela.
  - Ex: Trabalhador, Animal, TipoResultado, Vitoria …
- Essas classes podem ser qualificadas como classes abstratas, que no Java são identificadas com a palavra abstract na declaração

```
public abstract class Trabalhador { … }
public abstract class Vitoria extends TipoResultado { … }
```

- Essas classes podem conter atributos e métodos (inclusive construtores) , ser usadas em uma declaração polimórfica, mas não podem ser instanciadas

```
Vitoria v1 = new VitoriaPorPontos (30,27); // OK
Vitoria v1 = new Vitoria ( );  // ERRO!
```

# Métodos abstratos

- Assim como classes abstratas só servem de base para herança existem métodos que só são declarados na superclasse para serem sobrescritos nas subclasses
- Esses métodos podem ser qualificados como métodos abstratos e também usam a palavra reservada abstract na declaração
- Métodos abstratos não possuem corpo, apenas sua assinatura (cabeçalho) seguido de ponto-e-vírgula. Eles só podem ser declarados em classes abstratas.
- Construtores e métodos estáticos não podem ser abstratos

```
public abstract class Trabalhador {
    public abstract double getSalario ( ) ; …
public abstract class Vitoria {
    public abstract String toString ( ) ; …
```

# Classes concretas

- Chamamos as classes não abstratas de classes concretas, isto é as classes concretas que implementam todos os seus métodos
- Uma classe concreta que herdar, direta ou indiretamente, de uma classe abstrata deve OBRIGATORIAMENTE implementar todos os métodos abstratos de suas superclasses

```
public abstract class Horista extends Trabalhador {
    public double getSalario ( ) { // é obrigatório...

public class VitoriaPorNocaute extends Vitoria {
    // se não implementar é erro de compilação
    public String toString ( ) {
```

# Design por contrato

- A obrigatoriedade de implementação dos métodos abstratos nas classes concretas garantem um comportamento desejado
- Dessa forma, a superclasse abstrata representa uma espécie de contrato com uma interface uniforme entre as partes
  - Quem implementa a subclasse tem liberdade para adicionar ou modificar funcionalidades
  - Mas também tem a obrigação de definir certos comportamentos para poder usar métodos pré-definidos
- Esse modelo de desenvolvimento geralmente é chamado de design por contrato

# Exemplo

- Em um jogo os personagens tem sua vida determinada por um número inteiro de pontos de vida, além de um conjunto de atributos que determinam o seu poder de ataque e seu nível de defesa. Cada classe de personagem tem uma maneira específica de calcular o ataque e defesa, mas durante um movimento de ataque os pontos de vida de quem sofreu o ataque são diminuídos pela diferença entre o poder de ataque de quem está efetuando e o nível de defesa de quem está sofrendo. Como implementar a funcionalidade de ataque sem criar um método específico para cada classe ?

# Exemplo (cont.)

- O primeiro passo é criar uma classe abstrata de personagem que tenha métodos abstratos para calcular o ataque e defesa

```
public abstract class Personagem {
     private int pontosDeVida;
     private int []  atributos;

/* construtor, get e set */

     public void sofreDano (int dano){
          this.pontosDeVida -= dano;
     }

     public abstract int getPoderAtaque( ) ;
     public abstract int getNivelDefesa( ) .;
```

# Exemplo (cont.)

- Pode-se agora implementar o método de ataque mesmo sem saber de que classe é o personagem

```
public class Jogo {

    public static void ataque(Personagem a, Personagem d){
        int dano = a.getPoderAtaque() - d.getNivelDefesa();
        if(dano>0)
            d.sofreDano(dano);
    }
```

# Exemplo (cont.)

- Isso funciona porque todos os objetos das subclasses concretas de Personagem terão orbrigatoriamente os dois métodos abstratos implementados

```
public class Guerreiro extends Personagem {
     public int getPoderAtaque( ){
          return atributos[0]*atributos[1];
     }
     public int getNivelDefesa( ) {
          return atributos[1]+atributos[2];
     }
}

public class Mago extends Personagem {
     public int getPoderAtaque( ){
          return atributos[2]*atributos[3];
     } ...
```

# Exercícios

Refatorar as classes do exemplo das lutas para definir as classes e métodos abstratos plausíveis. Criar um cartel de lutas usando polimorfismo.