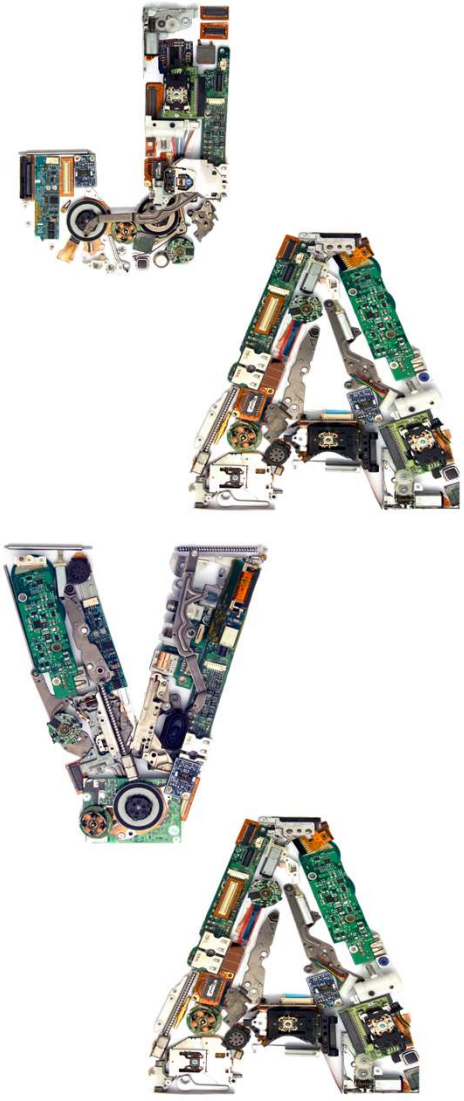
Programação Orientada a Objetos com Java e WEB

Polimorfismo



CONTEÚDO

Polimorfismo

1. Herança
2. Relacionamento TEM-UM
3. Polimorfismo
4. Classes abstratas
5. Métodos abstratos
6. Classe *Object*

1. Herança

- Quando você quiser saber se uma classe deve estender outra, aplique o teste: **É-UM**. Por exemplo:
  - O Triângulo É-UMA Forma!
  - O Gato É-UM Felino!
  - O Cirurgião É-UM Médico!
  - **Livro estende Autor. Faz sentido?? NÃO!!**

- Para saber se os tipos foram projetados corretamente faça a pergunta: "Faz sentido dizer X É-UM tipo Y?

- Livro e Autor estão relacionados, mas não através da herança (relacionamento **TEM-UM**).

1. Herança

- A relação entre Livro e Autor é **TEM-UM**, ou seja, a classe Livro tem uma variável de instância do tipo Autor.

- A relação **É-UM** é baseada na herança de classes ou implementação de interfaces enquanto que a relação **TEM-UM** é baseada na utilização.

1. Herança

- O teste **É-UM** funciona em qualquer local da árvore de herança.
- Se a classe B estende a classe A, ela **É-UMA** classe A.
- Se a classe C estender a classe B, ela passará no teste **É-UM** tanto com a classe B quanto com a classe A.
- A herança foi montada corretamente?
  - Tudo indica que se as classes passarem no teste **É-UM** elas farão parte da hierarquia de herança.
  - O relacionamento **É-UM** funciona apenas em uma direção, ou seja, um Triângulo é uma Forma, mas uma Forma não é um Triângulo.

1. Herança

- Não use herança se a superclasse e a subclasse não passarem no teste **É-UM.**
- Não use herança para simplesmente reutilizar o código de outra classe.

1. Herança

- O que pode ser herdado?
- A subclasse sempre herda ***membros*** da superclasse.
- ***Membros: variáveis de instâncias e métodos.***
- A superclasse pode selecionar o que uma subclasse poderá "visualizar" através do nível de acesso (encapsulamento) do membro.
- O nível de acesso controla "*quem vê o quê*".

Membros públicos são herdados e ficam visíveis, mas membros privativos São herdados, mas não ficam visíveis!

1. Herança

- Vantagens da herança:

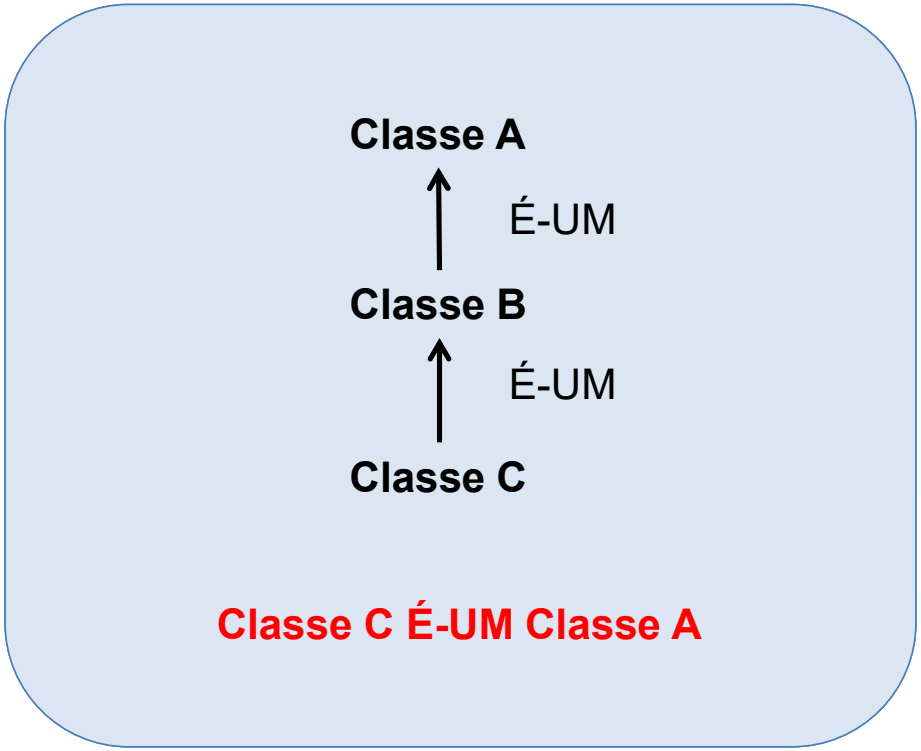
  - ***Eliminar código duplicado***;
  - Generaliza o comportamento comum a um grupo de classes, inserindo esse código na superclasse.
  - Alterações são feitas na superclasse apenas. Não há necessidade de alterar as subclasses.
  - A herança permite que todas as classes agrupadas sob um certo supertipo tenham todos os métodos que o supertipo tem.
  - ***Usar o polimorfismo.***

1. Herança – Resumo

- Uma subclasse estende a superclasse.
- A subclasse herda todos os membros públicos, mas não os privativos.
- Métodos herdados podem ser sobrepostos; as variáveis de instância não.
- ***Valide a hierarquia com o teste É-UM. Se X estende Y, então, X É-UM Y.***
- O teste **É-UM** funciona em apenas uma direção.

1. Herança – Resumo

- Se a classe B estende a classe A e C estende B, a classe B É-UMA classe A e a classe C É-UMA B, portanto, a classe C também É-UMA classe A;



2. Relacionamento TEM-UM

- É baseado na utilização, em vez de herança;
- Esse relacionamento significa que a classe A ***TEM-UM*** B, ou seja, o código da classe A tem uma *referência a uma instância* da classe B;
- Você pode dizer: Um Cavalo É-UM Animal. Um Cavalo TEM-UMA Rédea;

3. Polimorfismo – Introdução

- Imagine o sistema de um banco que deve controlar a entrada e a saída de seus funcionários. Observe a classe ***ControleDePonto***:

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class ControleDePonto {
    public void registraEntrada(Gerente g) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();
        System.out.println("ENTRADA: " + g.getCodigo());
        System.out.println("DATA: " + sdf.format(agora));
    }

    public void registraSaida(Gerente g) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();
        System.out.println("SAÍDA: " + g.getCodigo());
        System.out.println("DATA: " + sdf.format(agora));
    }
}
```

Fonte: K19 Treinamento → www.facebook.com/k19treinamento

3. Polimorfismo – Introdução

Secretários

Como controlar o ponto de todos funcionários?

Gerentes

Telefonistas

Fonte: K19 Treinamento → www.facebook.com/k19treinamento

3. Polimorfismo – Introdução

- Inicialmente podemos pensar em modelar o sistema usando herança.

```
public class Funcionario {
    private int codigo;

    //métodos da classe
}
```

```
public class Gerente extends Funcionario {
    private String usuario;
    private String senha;

    //métodos da classe
}
```

```
public class Telefonista extends Funcionario {
    private int ramal;

    //métodos da classe
}
```

Fonte: K19 Treinamento → www.facebook.com/k19treinamento

3. Polimorfismo – Introdução

- Além do reaproveitamento de código, a herança permite que todos os objetos sejam tratados como objetos da classe mais genérica (superclasse);

- Em outras palavras, um objeto da classe ***Gerente*** pode ser tratado como um objeto da classe ***Funcionario***, assim como um objeto da classe ***Telefonista*** também pode ser tratado como um objeto ***Funcionario***;

```
Gerente g = new Gerente();
Funcionario f = g;
```


Fonte: K19 Treinamento → www.facebook.com/k19treinamento

3. Polimorfismo – Introdução

- Melhorando o sistema de controle de ponto:

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class ControleDePonto {
    public void registraEntrada(Funcionario f) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();
        System.out.println("ENTRADA: " + f.getCodigo());
        System.out.println("DATA: " + sdf.format(agora));
    }

    public void registraSaida(Funcionario f) {
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        Date agora = new Date();
        System.out.println("SAÍDA: " + f.getCodigo());
        System.out.println("DATA: " + sdf.format(agora));
    }
}
```

Fonte: K19 Treinamento → www.facebook.com/k19treinamento

3. Polimorfismo

Polimorfismo: “Programar no geral ao invés do específico”.

Capacidade de tratar objetos criados a partir das classes específicas como objetos de uma classe genérica

Facilita a manutenção das classes

3. Polimorfismo

- Supondo que uma classe mais generalizada, Animal, possua o método mover, e que este método retorne a quantidade de metros percorrido a cada iteração.
  - Cada animal descendente da classe Animal, implementará de forma diferenciada o método mover;
  - A classe Peixe poderá mover 0,3 metros por iteração
  - A classe Pássaro poderá mover 0,8 metros por iteração;
  - Para incluir um novo animal não é necessário construir todas as definições novamente, basta utilizar a herança e implementar os comportamentos obrigatórios, como por exemplo, o método mover.

3. Polimorfismo

- O polimorfismo permite que uma variável de referência e o objeto sejam diferentes;
- No polimorfismo, o tipo de referência pode ser uma superclasse para o tipo do objeto real;
  - Qualquer coisa que estenda o tipo declarado para a variável de referência poderá ser atribuída a ela;
- Isso permite criar matrizes e vetores polimórficos;
- Com o polimorfismo, você pode escrever um código que não tenha que ser alterado quando novos tipos de subclasse foram introduzidos no programa.

3. Polimorfismo

- Qualquer objeto Java que passe no teste *É-UM* pode ser considerado polimórfico;
- Todos os objetos Java (com exceção dos objetos *Object*) são polimórficos;
- Objetos polimórficos podem ser acessados por meio de variáveis de referência;
- Uma variável de referência pode referenciar qualquer objeto do mesmo tipo que a referência declarada, *ou pode se referir a qualquer subtipo do tipo declarado*;

3. Polimorfismo

Uma variável de referência pode ser declarada como um tipo de classe ou um tipo de interface

Se a variável for de um tipo de interface, ela pode referenciar qualquer objeto de qualquer classe que implemente a interface

4. Classes abstratas

- *São classes que não podem ser instanciadas e tem o propósito único de atuar como uma superclasse;*

- Quando usar uma classe abstrata?
  - Quando a classe apresentar pelo menos um método abstrato.

- *Uma classe que não é abstrata é chamada de concreta*;

- Ao projetar uma hierarquia de herança você terá de decidir quais classes serão abstratas e quais serão concretas;

- *Uma classe abstrata pode ser utilizada como tipo de referência para fins polimórficos*;

FIAP

5. Métodos abstratos

- *Uma classe abstrata deve ser estendida;*
- *Um método abstrato significa que ele deve ser sobreposto;*
- Em algumas situações, você pode chegar a conclusão que alguns métodos não tem sentido ter código na superclasse. Tais métodos devem ser definidos como abstratos;

Um método abstrato não tem corpo!!

```
public abstract double calcularArea();
```

5. Métodos abstratos

- Exemplo:

```
public abstract double calcular(double x, double y);
```

- Caso você declare um método como abstrato deverá declarar a classe como abstrata também;

Não é possível ter um método abstrato em uma classe não-abstrata!

Implementar um método abstrato é sobrepor (override) o método

5. Métodos abstratos

Implementar um método abstrato é fornecer um corpo ao método;

- Eles existem somente por causa do polimorfismo;
- A primeira classe concreta da árvore de herança deve implementar todos os métodos abstratos;
- Ao implementar um método abstrato você deve criar em sua classe um método não-abstrato com a mesma assinatura (nome e argumentos) que seja compatível com o método abstrato da superclasse;

6. Classe *Object*

- ***É a mãe de todas as classes; é a superclasse de tudo!***

- Desde o início você esteve criando classes que eram subclasses da classe ***Object***;

- Toda classe que você criar estenderá ***Object,*** sem que seja preciso declará-lo;

- Qualquer classe que não estender explicitamente outra classe estenderá implicitamente ***Object***;

6. Classe *Object*

- Métodos da classe *Object*:
  - equals(Object o);
  - getClass();
  - hasCode();
  - toString();

6. Classe *Object* – método *toString*()

- A linguagem de programação Java fornece uma implementação padrão para o método ***toString*()** da classe ***java.lang.Object*** que é herdada por todas as classes Java;

- O retorno do método não é muito intuitivo. Seu retorno é composto pelo nome da classe seguido por um arroba (@) e pela representação do código ***hash*** em hexadecimal sem sinal;

```
public String toString() {
   return getClass().getName() + "@"
                + Integer.toHexString(hashCode());
}
```

6. Classe *Object* – método *toString*()

- O método ***toString*()** deve retornar uma representação informativa do objeto, que seja fácil de ler e entender. Fornecer uma boa implementação do método tornará a classe mais agradável de usar;

O método *toString*() é sempre chamado de forma automática quando nenhum método é especificado junto a variável de referência

- Para ser considerado como um método prático, o método ***toString*()** deve retornar todas as informações interessantes contidas no objeto;

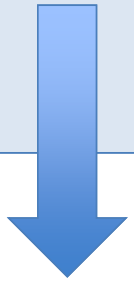
- O ideal é que a ***String*** sempre seja autoexplicativa.

6. Classe *Object* – método *toString*()

- Exemplo de utilização do método ***toString*()** original:

```
public class Aluno {
    private int rm;
    private String nome;

    public static void main(String[] args) {
        Aluno aluno = new Aluno();
        System.out.println(aluno);
    }
}
```



```
Aluno@15db9742
```

Quando nenhum método é explicitamente chamado junto a variável de referência, o método *toString*() será invocado.

6. Classe *Object* – método *toString*()

- Sobrescrevendo o método ***toString*()** original:

```
public class Aluno {
    private int rm;
    private String nome;

    public String toString() {
        String aux = "";
        aux += "RM -> "+rm+"\n";
        aux += "Nome -> "+nome+"\n";
        return aux;
    }

    public static void main(String[] args) {
        Aluno aluno = new Aluno();
        System.out.println(aluno);
    }
}
```

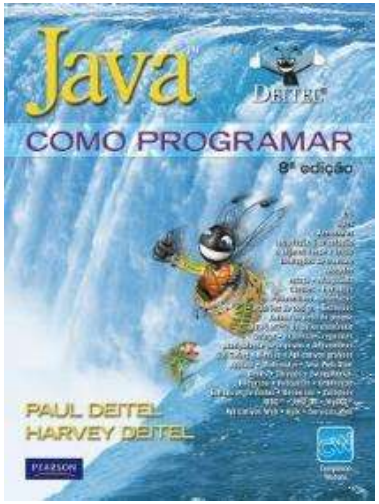
Saída do método *toString*() sobreposto:

```
RM -> 0
Nome -> null
```

REFERÊNCIAS

- DEITEL, H. M., DEITEL, P. J. **JAVA como programar**. 8ª edição. São Paulo: Prentice-Hall, 2010.

- SCHILDT, H. **Java para Iniciantes – Crie, Compile e Execute Programas Java Rapidamente**. 6ª Edição, Editora Bookman, Porto Alegre, RS, 2015.



REFERÊNCIAS

- KNUDSEN, J., NIEMEYER, P. **Aprendendo Java**. Rio de Janeiro: Editora Elsevier Campus, 2000.
- FLANAGAN, D. **Java – o guia essencial**. Porto Alegre: Editora Bookman, 2006.

REFERÊNCIAS

- ARNOLD, K., GOSLING, J., HOLMES, D., **Java programming language**. 4th Edition, Editora Addison-Wesley, 2005.

- JANDL JUNIOR, P. **Introdução ao Java**. São Paulo: Editora Berkeley, 2002.