Orientação a Objetos

Orientação a Objetos - Herança, Classes Abstratas e Polimorfismo

- A herança permite a uma classe herdar o estado (atributos) e o comportamento (métodos) de outra classe.

- A herança costuma ser utilizada quando existem, em diferentes classes, atributos e métodos semelhantes.

- Utilizando a herança, duas ou mais classes podem compartilhar alguns atributos ou métodos.

- Em um sistema para cálculo de folha de pagamento, vamos considerar a existência de dois tipos de funcionários:
  - Funcionário mensalista
    - Salário fixo mensal
  - Funcionário horista
    - Salário calculado em acordo com o número de horas trabalhadas.

- Apresentação da classe FuncionarioMensalista (os métodos get e set foram suprimidos somente para economizar espaço no slide):

```
class FuncionarioMensalista
{
   private string nome;
   private DateTime dataNascimento;
   private double salario;

   public double retornarSalario()
   {
       return salario;
   }
}
```

- Apresentação da classe FuncionarioHorista (os métodos get e set foram suprimidos somente para economizar espaço no slide):

```
class FuncionarioHorista
{
   private string nome;
   private DateTime dataNascimento;
   private double valorPorHora;
   private double horasTrabalhadas;

   public double retornarSalario()
   {
       return horasTrabalhadas * valorPorHora;
   }
}
```

- É possível identificarmos dois atributos comuns a FuncionarioMensalista e a FuncionarioHorista: nome e dataNascimento. Os métodos get e set para esses atributos também são iguais.

- Como FuncionarioMensalista e FuncionarioHorista são tipos de funcionários, podemos utilizar um recurso chamado herança e extrair os atributos e métodos comuns para uma classe que chamaremos de superclasse.

- Superclasse Funcionário: (os métodos get e set foram suprimidos somente para economizar espaço no slide):

```
class Funcionario
{
  private string nome;
  private DateTime dataNascimento;
}
```

- Agora a subclasse FuncionarioMensalista pode ser definida como (os métodos get e set foram suprimidos somente para economizar espaço no slide):

```
class FuncionarioMensalista : Funcionario
{
  private double salario;

  public double retornarSalario()
  {
      return salario;
  }
}
```

- Já a subclasse FuncionarioHorista pode ser definida como (os métodos get e set foram suprimidos somente para economizar espaço no slide):

```
class FuncionarioHorista : Funcionario
{
   private double valorPorHora;
   private double horasTrabalhadas;

   public double retornarSalario()
   {
       return horasTrabalhadas * valorPorHora;
   }
}
```

- Ao estender a classe Funcionario, a classe FuncionarioMensalista passa a também possuir todos os atributos e métodos da classe Funcionario.

- Diz-se que FuncionarioMensalista herda de Funcionario.

- Diz-se que a classe Funcionario é a superclasse ou classe pai, enquanto a classe FuncionarioMensalista é a subclasse ou classe filha.

- Exemplo de utilização da classe FuncionarioMensalista:

```
FuncionarioMensalista func1 = new FuncionarioMensalista();

func1.setNome(“José”);
func1.setDataNascimento(new DateTime(1995,1,1));
func1.setSalario(1000.00);

Console.Write(“Salário do ”);
Console.Write(func1.getNome());
Console.Write(“: ”);
Console.Write(func1.retornarSalario());
```

- Exemplo de utilização da classe FuncionarioHorista:

```
FuncionarioHorista func2 = new FuncionarioHorista();

func2.setNome(“Ana”);
func2.setDataNascimento(02-01-1995);
func2.setValorPorHora (50.00);
func2.setHorasTrabalhadas(22);

Console.Write(“Salário do(a) ”);
Console.Write(func2.getNome());
Console.Write(“: ”);
Console.Write(func2.retornarSalario());
```

- Note que tanto a classe FuncionarioMensalista quanto a classe FuncionarioHorista possuem o método retornarSalario, porém com implementações diferentes. Se eu for criar uma nova classe funcionário (FuncionarioDiarista para exemplificar) também será adicionado o método retornarSalario).

- Como eu posso fazer para que todas as classes que herdam da classe Funcionario tenham que obrigatoriamente implementar o método retornarSalario?

- A resposta à pergunta anterior é: crie um método abstrato chamado “retornarSalario” na classe Funcionario.

- Uma classe que possui um método abstrato também precisa ser abstrata.

- Nova superclasse abstrata Funcionario: (os métodos get e set foram suprimidos somente para economizar espaço no slide):

```
abstract class Funcionario
{
   private string nome;
   private DateTime dataNascimento;
   public abstract double retornarSalario();
}
```

- Note que um método abstrato não possui implementação. Somente o seu tipo de dados, nome e parâmetros recebidos devem ser definidos. Essas informações são conhecidas como “assinatura do método”.

- Nova subclasse FuncionarioMensalista (os métodos get e set foram suprimidos somente para economizar espaço no slide):

```
class FuncionarioMensalista : Funcionario
{
  private double salario;

  public override double retornarSalario()
  {
      return salario;
  }
}
```

- Nova subclasse FuncionarioHorista (os métodos get e set foram suprimidos somente para economizar espaço no slide):

```
class FuncionarioHorista : Funcionario
{
   private double valorPorHora;
   private double horasTrabalhadas;

   public override double retornarSalario()
   {
       return horasTrabalhadas * valorPorHora;
   }
}
```

- Uma característica das classes abstratas é que as mesmas não podem ser instanciadas. Anteriormente, era possível fazer algo como:

```
Funcionario func1 = new Funcionario();
```

- Como a classe Funcionario agora é abstrata, já não é mais possível instanciar objetos dessa classe.
  - Para esse caso, esse comportamento é desejado e útil!

- Polimorfismo é o nome dado quando uma operação de uma superclasse é implementada de maneira diferente por duas ou mais de subclasses.
  - Essas implementações diferentes fazem com que um mesmo método chamado de um objeto do tipo da superclasse tenha comportamentos diferentes.

- Polimorfismo é o conceito que descreve a capacidade de um tipo A ser usado como um tipo B.

- O objetivo do polimorfismo é implementar um estilo de programação baseado em passagem de mensagens no qual objetos de diferentes tipos definem uma mesma interface de operações.

- A herança é o princípio da orientação a objetos que torna possível o mecanismo do polimorfismo.

- Para o exemplo das classes Funcionario, FuncionarioMensalista e FuncionarioHorista, podemos utilizar o seguinte recurso:

```
FuncionarioMensalista func1 = new FuncionarioMensalista();

func1.setNome(“José”);
func1.setDataNascimento(01-01-1995);
func1.setSalario(1000.00);

Funcionario funcGenerico = func1;

Console.Write(“Salário do ”);
Console.Write(funcGenerico.getNome());
Console.Write(“: ”);
Console.Write(funcGenerico.retornarSalario());
```

- Só foi possível utilizar a linha

```
Funcionario funcGenerico = func1;
```

devido ao objeto filho (instanciado à partir da subclasse) poder ser utilizado no lugar do objeto pai (definido na superclasse).

- Vejamos o mesmo caso anterior, porém para a classe FuncionarioHorista:

```
FuncionarioHorista func2 = new FuncionarioHorista();

func2.setNome(“Ana”);
func2.setDataNascimento(02-01-1995);
func2.setValorPorHora (50.00);
func2.setHorasTrabalhadas(22);

Funcionario funcGenerico = func2;

Console.Write(“Salário do(a) ”);
Console.Write(funcGenerico.getNome());
Console.Write(“: ”);
Console.Write(funcGenerico.retornarSalario());
```

- Veja novamente as seguintes linhas:
  - `Funcionario funcGenerico = func1;`
    - func1 sendo uma instância de FuncionarioMensalista
  - `Funcionario funcGenerico = func2;`
    - func2 sendo uma instância de FuncionarioHorista
- Note que ao executar o método retornarSalario em cada um dos objetos Funcionario, o comportamento do método será diferente.
  - Isso ocorre devido ao Funcionario ser, na verdade, um FuncionarioMensalista ou um FuncionarioHorista, mas quem está utilizando o objeto, o enxerga apenas como sendo um Funcionario.
  - **Isso é polimorfismo!**

- Agora vamos explorar um pouco mais o poder do polimorfismo:
  - Imagine que você possui uma folha de pagamento contendo um vetor de Funcionario
    - Um funcionário pode ser FuncionarioMensalista ou FuncionarioHorista).
  - Devido ao polimorfismo, mesmo sem saber se um determinado Funcionario é um FuncionarioHorista ou FuncionarioMensalista, você consegue calcular o valor total a ser pago adequadamente utilizando o pseudocódigo do próximo slide:

```
private double totalFolhaPagamento = 0;

for (int i = 0; i < vetorFuncionarios.Tamanho; i++)
{
    totalFolhaPagamento += vetorFuncionarios[i].retornarSalario();
}

Console.Write("Soma dos pagamentos de todos os funcionários: ");
Console.Write(totalFolhaPagamento);

//Isso é polimorfismo!
```

Outros conceitos de OO serão apresentados quando forem mostrados conceitos avançados do Diagrama de Classes.