Aula 05 – POO (Programação Orientada a Objetos)

1. Introdução [i]

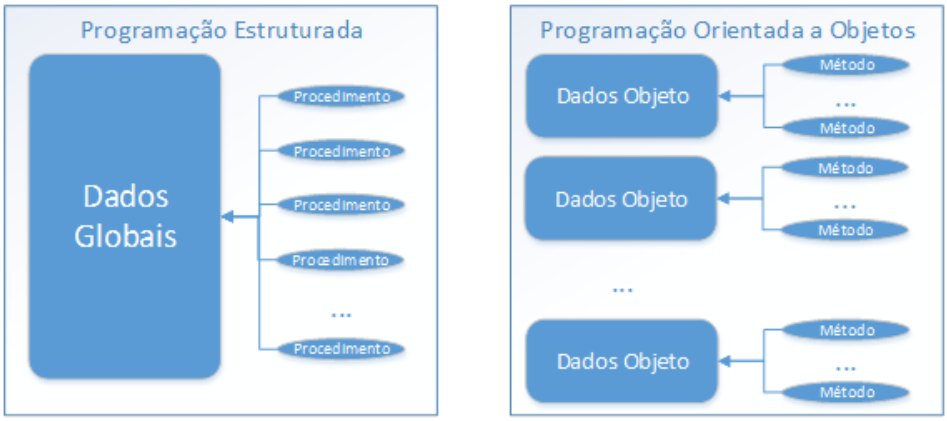
Programação orientada a objetos é um estilo de programação que permite os desenvolvedores agruparem tarefas semelhantes em classes. Isso ajuda a mantermo-nos dentro do princípio "*don't repeat yourself*" (*DRY*) (em português, não se repita), além de facilitar a manutenção do código.

> **"Programação orientada a objetos é um estilo de programação que permite os desenvolvedores agruparem tarefas semelhantes em classes."**

Um dos maiores benefícios da programação DRY é que, se alguma informação é alterada em seu programa, geralmente, só uma mudança é necessária para atualizar o código.

1.1 Programação Estruturada vs Programação Orientada a Objetos

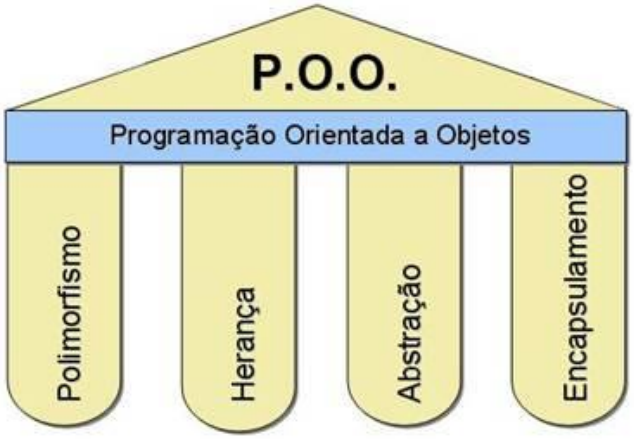
Na programação estruturada, temos procedimentos (ou funções) que são aplicados globalmente em nossa aplicação, na orientação a objetos, temos métodos que são aplicados aos dados de cada objeto. Essencialmente, os procedimentos e métodos são iguais, sendo diferenciados apenas pelo seu escopo.



Fonte: http://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264

1.2 Pilares da POO

Os requerimentos de uma linguagem para ser considerada nesse paradigma, precisa atender a quatro tópicos bastante importantes:



Fonte: http://www.devmedia.com.br/artigo-clube-delphi-93-orientacao-a-objetos-no-delphi-for-php/10635

1.2.1 Abstração

A abstração consiste em um dos pontos mais importantes dentro de qualquer linguagem Orientada a Objetos, como estamos com uma representação de um objeto real, devemos prever o que esse objeto irá realizar dentro de nosso sistema, observando:

- **Identidade:** Deve ser única dentro do sistema;
- **Características do objeto (atributos):** São os elementos que o definem, essas características são nomeadas propriedades. Por exemplo, as propriedades de um objeto "Carro" poderiam ser "cor", "ano" e "modelo".
- **Métodos:** As ações que o objeto irá executar.

1.2.2 Encapsulamento

Um dos elementos que adicionam segurança à aplicação em uma programação orientada a objetos pelo fato de esconder as propriedades, criando uma espécie de caixa preta baseada em propriedades privadas, ligadas a métodos especiais chamados getters e setters, que irão retornar e setar o valor da propriedade, respectivamente, evitando assim o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação.

1.2.3 Herança

O reuso de código é uma das grandes vantagens da programação orientada a objetos, essa característica otimiza a produção da aplicação em tempo e linhas de código.

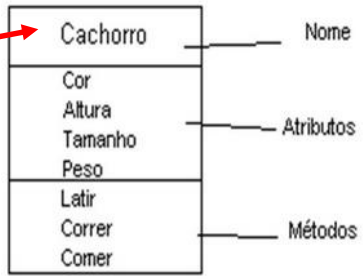
1.2.4 Polimorfismo

Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Os objetos filhos herdam as características e ações, mas em alguns casos, é necessário que as ações para um mesmo método seja diferente, assim ocorre o polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

2. Classes

A sintaxe para criar uma classe é bem direta: declare-a usando a palavra-chave ***class***, seguida do nome da classe e um par de chaves ({}). A classe é uma estrutura que abstrai um conjunto de objetos contendo características similares, uma classe define o comportamento de seus objetos usando métodos e modificando seus estados como os atributos, exemplo:

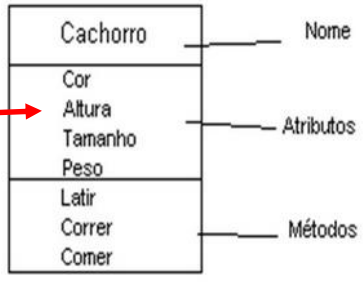
```
<?php
    class Cachorro {
        // estrutura da classe
    }
?>
```



2.1 Propriedades da Classe (Atributos)

Para adicionar dados à classe, usamos as propriedades, que são variáveis específicas à classe e só podem ser acessadas usando o objeto.

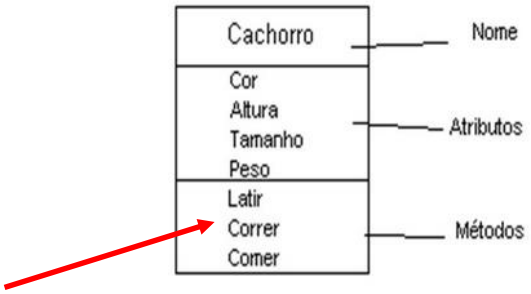
```
<?php
    class Cachorro {
        public $cor;
        public $altura;
        public $tamanho;
        public $peso;
    }
?>
```



2.2 Métodos de Classe

Métodos são funções específicas das classes. Ações particulares que os objetos serão capazes de executar e são definidas dentro das classes.

```php
<?php
        class Cachorro {
                public   $cor;
                public   $altura
                public   $tamanho
                public   $peso
                function latir(){    }
                function correr(){    }
                function comer(){    }
        }
?>
```



2.3 Criando instância

Para criar uma instância de uma classe, a instrução **new** deve ser utilizada. Um objeto sempre será criado a não ser que a classe tenha um construtor definido que dispare uma exceção em caso de erro. Classes devem ser definidas antes de instanciadas (e em alguns casos isso é obrigatório).

```php
<?php
        //criando instância da Classe Cachorro
        $objCachorro = new Cachorro();
?>
```

2.4 – Manipulando dados

Para manipular os dados das propriedades devemos levar em consideração a visibilidade desta propriedade: ***public*** ou ***private***.

2.4.1 – Dados tipo public

Para acesso, a dados do tipo ***public*** é direto após a instância do objeto.

```php
<?php
   class Cachorro {
      public $nome;
   }
```

```
        echo "Manipulando dados<br><br>";
        // instanciando a classe
        $objCachorro = new cachorro();
        // atribuindo valor para a propriedade nome
        $objCachorro->nome = "Dog";
        // imprimindo o valor da propriedade
        echo "O nome do cachorro é ".$objCachorro->nome;
?>
```

Arquivo: exemplo01.php

2.4.1 – Dados tipo private

Para dados do tipo ***private***, utilizaremos os métodos especiais chamados getters e setters.

```
<?php
    class Cachorro {
        private $nome;
        public function setNome( $valor ){
            $this->nome = $valor;
        }
        public function getNome(){
            return $this->nome;
        }
    }
    echo "Manipulando dados - GET e SET <br><br>";
    // instanciando a classe
    $objCachorro = new cachorro();
    // atribuindo valor para a propriedade nome
    $objCachorro->setNome("Dog");
    // imprimindo o valor da propriedade
    echo "O nome do cachorro é ".$objCachorro->getNome();
?>
```

Arquivo: exemplo02.php

Podemos criar funções genéricas para os métodos GET() e SET().

```
<?php
    class Cachorro {
        private $nome;
                                                private $cor;
        public function __set($atributo, $valor){
            $this->$atributo = $valor;
```

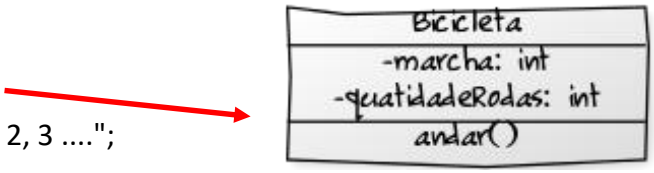
```
    }
    public function __get($atributo){
        return $this->$atributo;
    }
}
echo "Manipulando dados - GET e SET <br><br>";
// instanciando a classe
$objCachorro = new cachorro();
// atribuindo valor para a propriedade nome
$objCachorro->nome = "Dog";
$objCachorro->cor = "Preta";
// imprimindo o valor da propriedade
echo "O nome do cachorro é ".$objCachorro->nome;
echo ", e ele tem a cor ".$objCachorro->cor;
?>
```

Arquivo: exemplo03.php

3. Classes – Herança

Observe a classe a seguir:

```
<?php
class Bicicleta {
    private $marcha;
    private $quantidadeRodas;
    public function passarMarcha() {
        echo "Mudando a marcha .... 1, 2, 3 ....";
    }
    public function andar() {
        echo "Estou andando ... cai!!!";
    }
    // instanciando objeto
    $objBiclicleta = new Bicicleta();
    // executando método
    $objBiclicleta->andar();
}
?>
```



Arquivo: exemplo04.php

Criando a classe ***Bike***, que recebe por herança os atributos e métodos da classe ***Bicicleta***.

```
<?php
class Bicicleta {
	private $marcha;
	private $quantidadeRodas;
	public function passarMarcha() {
		echo "Mudando a marcha .... 1, 2, 3 ....";
	}
	public function andar() {
		echo "Estou andando ... cai!!!";
	}
}
// definição da classe Bike
class Bike extends Bicicleta
{

}
// instanciando objeto
$objBiclicleta = new Bike();
// executando método
$objBiclicleta->andar();
?>
```

Arquivo: exemplo05.php

Exemplo: Executando método da classe filho (Bike).

```
<?php
class Bicicleta {
	private $marcha;
	private $quantidadeRodas;
	public function passarMarcha() {
		echo "Mudando a marcha .... 1, 2, 3 ....";
	}
	public function andar() {
		echo "Estou andando ... cai!!!";
	}
}
// definição da classe Bike
class Bike extends Bicicleta
{
	public function parar() {
		echo "Pare a bicleta para não cair.<br>";
```

```
    }
}
// instanciando objeto
$objBiclicleta = new Bike();
// executando método
$objBiclicleta->parar();      // método da classe pai
$objBiclicleta->andar();      // método da classe filho
?>
```

Arquivo: exemplo06.php

4. Classes – Polimorfismo

Observe a classe a seguir, onde o método ***calculaJuros*** realiza o cálculo dos juros.

```
<?php
class DividaBanco
{
        private $valor;
        public function setValor($valor){
                $this->valor = $valor;
        }
        public function getValor(){
                return $this->valor;
        }
        public function calculaJuros() {
                return $this->valor * 1.50;
        }
}
$objDividaBanco = new DividaBanco();
// passagem de valor
$objDividaBanco-> setValor(1000);
echo "Sua dívida com o Banco é: R$ ";
echo $objDividaBanco->calculaJuros();
?>
```

Arquivo: exemplo07.php

A classe ***DividaMae***, possui o mesmo método da classe ***DividaBanco***, lembrando da definição anterior:

> "Os objetos filhos herdam as características e ações, mas em alguns casos, é necessário que as ações para um mesmo método seja diferente, assim ocorre o

polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai."

O resultado do método ***calculaJuros***, é a multiplicação do valor por 0.5, assim o método filho sobrepõe o método pai.

```
<?php
 class DividaBanco
        {
        private $valor;
        public function setValor($valor){
                $this->valor = $valor;
        }
        public function getValor(){
                return $this->valor;
        }
        public function calculaJuros() {
                return $this->valor * 1.50;
        }
}
// criando classe
class DividaMae extends DividaBanco
{
        public function calculaJuros()
        {
                return parent::getValor() * 0.5;
        }
}
$objDividaMae = new DividaMae();
// passagem de valor
$objDividaMae->setValor(1000);
echo "Sua dívida com o Banco é: R$ ";
echo $objDividaMae->calculaJuros();
?>
```

Arquivo: exemplo08.php

4. Classes – Executando Operações

Podemos criar classes somente com métodos para executar determinadas operações, no exemplo a seguir, o método ***calcular*** realiza a operação com valores fixos.

```php
<?php
    class matematica {
            public function calcular(){
                    return 10 * 5;
            }
    }
echo "Manipulando dados<br><br>";
// instanciando a classe
$obj_mat = new matematica();
echo "A multiplicação de 10 x 5 =  ". $obj_mat->calcular();
?>
```

Arquivo: exemplo09.php

Melhorando o método, agora os valores por parâmetros.

```php
<?php
    class matematica {
          public function calcular($a, $b){
                  return $a * $b;                    // $a e $b representam os valores $x e $y na ordem
          }
    }
echo "Manipulando dados<br><br>";
// instanciando a classe
$obj_mat = new matematica();
$x = 10;
$y = 5;
echo "A multiplicação de $x x $y =  ". $obj_mat->calcular($x,$y);
?>
```

Arquivo: exemplo10.php

4. Classes – Somente com Atributos

Da mesma forma que temos classes somente com métodos, podemos ter classes somente com atributos.

Observe que os valores já estão atribuídos e protegidos (***private***).

```php
<?php
    class Config {
      private $drive = "C";
      private $pasta = "Backup";
```

```
    public function __get($atributo){
        return $this->$atributo;
    }
  }
echo "Manipulando dados<br><br>";
$configuracao = new Config();
echo "O drive <b>".$configuracao->drive.":</b> está destinado para ".$configuracao->pasta;
?>
```

Arquivo: exemplo11.php

Pensando em banco de dados, temos:

```
<?php
class Conectar{
        private $host = "localhost";
        private $user = "root";
        private $pass = "usbw";
        private $banco = "banco";
        public function __get($atributo){
                return $this->$atributo;
        }
}
$conexao = new Conectar();
echo "<b>Servidor:</b> ".$conexao->host."<br>";
echo "<b>Usuário:</b> ".$conexao->user."<br>";
echo "<b>Senha:</b> ".$conexao->pass."<br>";
echo "<b>Banco de Dados:</b> ".$conexao->banco."<br>";
?>
```

Arquivo: exemplo12.php

Melhorando a classe de conexão.

```
<?php
class BancoDados{
        // dados de conexão do banco
        private $host = "localhost";
        private $user = "root";
        private $pass = "usbw";
        private $banco = "banco";
        // variável de conexão
        public $con;
```

```
// função para conexão com o Banco de dados
function conecta(){
        $this->con = @mysqli_connect($this->host,$this->user,$this->pass,$this->banco);
        // Conecta ao Banco de Dados
        if(!$this->con){
                // Caso ocorra um erro, exibe uma mensagem com o erro
                die ("Problemas com a conexão: ".mysqli_connect_error($this->con));
        } else {
                echo "Conexão realizada com Sucesso!!!!";
        }
}
}
$conexao = new BancoDados();
$conexao->conecta();
?>
```

Arquivo: exemplo13.php

Além da função ***mysqli_connect()***, utilizou-se também a função ***mysqli_connect_error()*** a qual retorna os erros no processo de conexão.

http://php.net/manual/pt_BR/mysqli.connect-error.php

[i] https://code.tutsplus.com/pt/tutorials/object-oriented-php-for-beginners--net-12762