**ANOTAÇÕES PROGRAMAÇÃO ORIENTADA A OBJETOS – PHP**

* **Evolução das linguagens de programação:**

Tudo foi evoluindo de programação de baixo nível até a POO de hoje em dia. Quem criou a POO foi o Alan Kay, trazendo mais próximo a realidade a programação para que facilite o trabalho e entendimento. O Alan trabalhou no setor de estudos da Xerox. O Alan também criou o SmallTalk já com conceitos de classes, métodos, atributos, etc.

Da Xerox ele foi para a Apple, Disney e na HP.

O que é um objeto? Coisa material ou abstrata que pode ser percebida pelos sentidos e descrita por meio das suas características, comportamentos e estado atual.

Vamos instanciar com os elementos abaixo, gerar uma classe em forma de objeto, isso é instanciar. Veja abaixo.

Exemplo:

Classe: Caneta;

Modelo: Caractere;

Cor: Caractere;

Ponta: Real;

Carga: Inteiro;

Tampada: Lógico;

**Método** rabiscar()

Se (tampada) então

Escreva(“ERRO”)

Senão

Escreva(“Rabisco”)

FimSe

**FimMetodo**

**Método** tampar()

Tampada = verdadeiro

**FimMetodo**

FimClasse

Com uma caneta eu tenho o seguintes:

* Atributos:

Modelo, Cor, Ponta, Carga,Tampada;

* Métodos:

Escrever, Rabiscar, Pintar, Tampar, Destampar

* Estado:

Definido em um momento específico com base nos atributos e métodos.

* **O que é visibilidade em um objeto?**

O que é Linguagem de Modelagem Unificada (LMU – UML)? Diagrama de classe:

Faça um retângulo e coloque o nome da classe em negrito o nome da classe no topo do retângulo. No meio coloque todos os atributos (características) desta classe, e na parte de baixo coloque todos os métodos da classe. Por exemplo:

|  |
| --- |
| **Caneta**  + modelo  + cor  - ponta  # carga  # tampada |
| + escrever()  + rabiscar()  + pintar()  - tampar() |

**Os modificadores de visibilidade indicam o nível de acesso aos componentes internos de uma classe (atributos e métodos). A Visibilidade indica o nível de acesso que temos até eles com os três símbolos (+ pública, - privado, # protegido).**

Assim eu dou acesso público, privado ou protegido para cada atributo ou método da minha classe.

Exemplos do dia a dia:

* Orelhão: acesso público;
* Celular: acesso privado;
* Telefone casa: protegido só para minha família usar.

Um atributo ou método público a classe atual e todas as outras classes tem acesso.

Um atributo ou método privado somente a classe atual pode ter acesso;

Um atributo ou método protegido só minha classe e seus filhos podem ter acessos.

Exemplo agora com a nossa classe Caneta e suas visibilidades:

**Class Caneta:**

Público modelo: Caractere;

Público cor: Caractere;

Público ponta: Real;

Protegido carga: Inteiro;

Protegido tampada: Lógico;

Público Metodo escrever()

Fim do Metodo

Público Metodo rabiscar()

Fim do Metodo

**FimClass**

C1 = nova Caneta

C1.modelo = “BIC cristal”

C1.escrever()

O que define se o atributo ou método da classe poderá ser alterado será a visibilidade, se é público, privado ou protegido. Cria-se meio que uma “cápsula” para proteger métodos e atributos que podem causar um não funcionamento da classe.

Para definir um novo valor e ter acesso a um atributo ou método privado ou protegido existem **Métodos Acessores e Métodos Modificadores:**

**Ou seja, para ter acesso ao privado ou protegido de um atributo ou método da classe, você precisará de um intermediário, que será o método público.**

* **Métodos Especiais:**

Você pode criar métodos com os nomes que quiser, porém existem padrões seguidos pelos programadores.

**Métodos Acessores (Getters)** são os métodos que dão acessos, pegam alguma informação até você. Você o identifica pelo Get. Por Exemplo: Saber o saldo da minha conta, “Get Saldo”.

Com isso, colocamos uma função intermediária que tem autorização para os dados para transmitir esse dado ao solicitante com o Get.

Por exemplo:

e = nova **Estante**

t = e.getTotDoc()

**Métodos Modificadores (Setters)** modificam coisas que estão dentro do objeto, sem deixar que diretamente mecham nos atributos do objeto, ele é o intermediador.

Por exemplo:

e = nova **Estante**

e.setTotDoc(**doc**).

Normalmente o método set precisa de um parâmetro para adicionar algo dentro do objeto.

Todos esses métodos especiais são públicos.

Exemplo prático:

Classe **Caneta**

Publico modelo: Caracetere

Privado ponta: Real;

Publico Método getModelo()

Retorne Modelo

FimMetodo

Publico Método setModelo(m:caractere)

Modelo = m

FimMetodo

**FimClasse**

**C1 = nova Caneta**

**C1.setModelo(“BIC Cristal”)**

**Escreva(c1.getModelo())**

**Método Construtor (Construct)** é um método especial que ele deixa com que após criar um novo objeto ele já inicie com parâmetro já estabelecidos. Por exemplo, objeto Caneta já criado tampada.

Método construtor sem parâmetros:

Classe **Caneta**

Metodo construtor ()

Tampar()

cor = “Azul”

FimMetodo

FimClasse

Método construtor com parâmetros:

Classe Caneta

Metodo construtor (m:Caractere, c: Caractere, p:Real)

setModelo(m)

setCor(c)

setPonta(p)

tampar()

FimMetodo

FimClasse

C1 = nova Caneta

(“BIC”, “Azul”, 0.5)

Ou seja, temos dentro de uma classe, seus atributos que lhe indicam suas características. Temos também seus métodos que indicam suas funções, que podem ser métodos especiais que são os famosos Getters e Setters e Constructor (Pegar infos, adicionar infos e atributos e valores padrões no ínicio do clase, respectivamente). Há também os métodos normais que são as funções que uma classe possa fazer.

Segue abaixo um exemplo de uma classe com seus métodos, métodos especiais e atributos:

<?php

class ContaBanco {

    //Atributos

    public $numconta;

    protected $tipo;

    private $dono;

    private $saldo;

    private $status;

    //Métodos

    public function AbrirConta($t) {

        $this->setTipo($t);

        $this->setStatus(true);

        if ($t == "CC") {

            $this->setSaldo(50);

        } elseif ($t == "CP") {

            $this->setSaldo(150);

        }

    }

    public function FecharConta() {

        if ($this->getSaldo() > 0) {

            echo "<p>Conta ainda tem dinheiro! Não posso fechá-la.</p>";

        } elseif ($this->getSaldo() < 0) {

            echo "<p> Você está em débito!</p>";

        } else {

            $this->setStatus(false);

        }

    }

    public function depositar($v) {

        if ($this->getStatus()) {

            $this->setSaldo($this->getSaldo() + $v);

        } else {

            echo "<p>Conta fechada, não consigo depositar!</p>";

        }

    }

    public function sacar($v) {

        if($this->getStatus()) {

            if ($this->getSaldo() > $v) {

                $this->setSaldo($this->getSaldo() - $v);

                echo "<p> Saque de $v autorizado na conta de  " .$this->getDono() . "</p>";

            } else {

                echo "<p> Saldo insuficiente para saque </p>";

            }

        } else {

            echo "<p> Não posso sacar de uma conta fechada </p>";

        }

    }

    public function pagarMensal() {

        if ($this->getTipo() == "CC") {

            $v = 12;

        }elseif ($this->getTipo() == "CP") {

            $v = 20;

        }

        if ($this->getStatus()) {

            $this->setSaldo($this->getSaldo() - $v);

        } else {

            echo "<p> Problemas com a conta, não posso cobrar!</p>";

        }

    }

    //Métodos especiais

    public function \_\_construct() {

        $this->setSaldo(0);

        $this->setStatus(false);

        echo "<p>Conta construída com sucesso!</p>";

    }

    public function getnumConta() {

        return $this->numconta;

    }

    public function setnumConta($n) {

        $this->numconta = $n;

    }

    public function getTipo() {

        return $this->tipo;

    }

    public function setTipo($t) {

        $this->tipo = $t;

    }

    public function getDono() {

        return $this->dono;

    }

    public function setDono($d) {

        $this->dono = $d;

    }

    public function getSaldo() {

        return $this->saldo;

    }

    public function setSaldo($sal) {

        $this->saldo = $sal;

    }

    public function getStatus() {

    return $this->status;

    }

    public function setStatus($sta) {

        $this->status = $sta;

    }

}

* **Pilares da POO:**

Existe os três pilares da POO: E – encapsulamento; H – herança; P- polimorfismo;

Ou seja, com isso podemos encapsular, temos heranças e várias formas na POO.

Um software encapsulado tem um padrão, o programador está protegido do software e vice versa, para que não haja problemas entre o programador e o software. Ou seja, ocultar partes da implementação permitindo construir partes invisíveis ao mundo exterior. O importante é a funcionalidade do software.

Na POO você pode conversar com o objeto, sua capsula, o que chamamos de mensagens. O mundo externo não entra na cápsula, porém conversa com a cápsula. Com isso ligamos o conceito de visibilidade e cápsula. Por exemplo: um atributo privado não há interação com o externo, mas só receberá mensagens e enviará retornos externos. Um bom objeto encapsulado tem uma boa interface (o que fornece comunicação com o mundo externo).

Se programar com encapsulamento ajuda na reutilização do código.

**Quando se encapsula se coloca todos os atributos como privados, ou até protegidos, mas nunca públicos.**

**Assim, coloco todos os métodos públicos da interface dentro da minha classe, além dos métodos especiais que estarão privados. Exemplo:**

|  |
| --- |
| **ControleRemoto**  - volume  - ligado  - tocando |
| + ligar()  + desligar()  + abrirMenu()  + fecharMenu()  +maisVolume()  +menosVolume()  +ligarMudo()  +desligarMudo()  +play()  +pause()  -setVolume()  -getVolume()  -setLigado()  - getLigado()  -setTocando()  -getTocando() |

Mas antes a gente digita os códigos da interface, criamos a interface para depois implementar ela a nossa classe. Quando colocamos em um método **“public abstract function”** dizemos que ele não será desenvolvido no mesmo lugar e sim em outro:

<?php

interface Controlador {

    public  function ligar();

    public  function desligar();

    public  function abrirMenu();

    public  function fecharMenu();

    public  function maisVolume();

    public  function menosVolume();

    public  function ligarMudo();

    public  function desligarMudo();

    public  function play();

    public  function pause();

}

Agora nós vamos desenvolver nossa classe, após termos desenvolvido nossa interface:

<?php

class ControleRemoto {

    //atributos

    private $volume;

    private $ligado;

    private $tocando;

    //metodos

    function \_\_constructor() {

        $this->volume = 50;

        $this->ligado = false;

        $this->tocando = false;

    }

    //metodos especiais

    function getVolume() {

        return $this->volume;

    }

    function getLigado() {

        return $this->ligado;

    }

    function getTocando() {

        return $this->tocando;

    }

    function setVolume($v) {

         $this->volume = $v ;

    }

    function setLigado($l) {

         $this->ligado = $l;

    }

    function setTocando($t) {

        return $this->tocando = $t;

    }

}

Agora vou implementar minha interface com meus métodos abstratos na minha classe:

<?php

    require\_once 'interfacecontrolador.php';

    class ControleRemoto implements Controlador {

    //atributos

    private $volume;

Meu código final:

INDEX.PHP:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=<device-width>, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1> Projeto Controle Remoto! </h1>

   <?php

   require\_once 'controle.php';

   $primeirocontrole = new ControleRemoto ();

   $primeirocontrole->ligar();

   $primeirocontrole->maisVolume();

   $primeirocontrole->maisVolume();

   $primeirocontrole->abrirMenu();

   $primeirocontrole->maisVolume();

   $primeirocontrole->maisVolume();

   $primeirocontrole->abrirMenu();

   print\_r($primeirocontrole);

   ?>

</body>

</html>

CONTROLEREMOTO.PHP:

<?php

    require\_once 'interfacecontrolador.php';

    class ControleRemoto implements Controlador {

    //atributos

    private $volume;

    private $ligado;

    private $tocando;

    //metodos

    function \_\_constructor() {

        $this->volume = 50;

        $this->ligado = false;

        $this->tocando = false;

    }

    public  function ligar() {

        $this->setLigado(true);

    }

    public  function desligar() {

        $this->setLigado(false);

    }

    public  function abrirMenu() {

        echo "<br> Está ligado?: ". $this->getLigado() ? "SIM <br>" : "NÃO <br>";

        echo "<br> Está tocando?: ". $this->getTocando() ? "SIM <br>" : "NÃO <br>";

        echo "<br> Volume: " .$this->getVolume();

        for ($i=0;  $i <= $this->getVolume(); $i+=10) {

            echo "|";

        }

        echo "<br>";

    }

    public  function fecharMenu() {

        echo "<br> Fechando menu...";

    }

    public  function maisVolume() {

        if  ($this->getLigado()) {

            $this->setVolume($this->getVolume() + 5);

        }

    }

    public  function menosVolume(){

        if ($this->getLigado()) {

        $this->setVolume($this->getVolume() - 5);

        }

    }

    public  function ligarMudo(){

        if ($this->getLigado() && $this->getVolume() >0) {

        $this->setVolume(0);

        }

    }

    public  function desligarMudo(){

        if ($this->getLigado() && $this->getVolume() <= 0) {

        $this->setVolume(50);

        }

    }

    public  function play(){

        if ($this->getLigado() && !$this->getTocando()) {

            $this->setTocando(true);

        }

    }

    public  function pause(){

        if ($this->getLigado() && $this->getTocando()) {

            $this->setTocando(false);

        }

    }

    //metodos especiais

    function getVolume() {

        return $this->volume;

    }

    function getLigado() {

        return $this->ligado;

    }

    function getTocando() {

        return $this->tocando;

    }

    function setVolume($v) {

         $this->volume = $v;

    }

    function setLigado($l) {

         $this->ligado = $l;

    }

    function setTocando($t) {

        return $this->tocando = $t;

    }

}

* **Relação de Diversas Classes:**

Criamos diversas classes com seus atributos e métodos normais e especiais. Lembre-se de encapsular as classes (métodos como privados, funções públicas).

|  |  |
| --- | --- |
| **LUTADOR** | |
|  |  |
| - | nome |
| - | nacionalidade |
| - | idade |
| - | altura |
| - | peso |
| - | categoria |
| - | vitórias |
| - | derrotas |
| - | empates |
| + | apresentar() |
| + | status() |
| + | ganharLuta() |
| + | perderLuta() |
| + | empatarLuta() |

Agora nós criamos um vetor com todos nossos lutadores que possuirão estes atributos e funções. Ou seja, um vetor de lutadores da mesma classe:

**L: vetor[0,1,2,3,4,5]**

**L[0] = new Lutador(“Pretty Boy”, “França”, 31, 1.75, 68.9, 11, 3, 1)**

**L[1] = new Lutador(...)**

**L[2] = new Lutador(...)**

**L[3] = new Lutador(...)**

**L[4] = new Lutador(...)**

**L[5] = new Lutador(...)**

Para realizar as funções com os lutadores do vetor após a atribuição dos valores dos atributos, fazemos:

**L[0].apresentar()**

**L[1].lutar()**

Para fazermos a relação de classes, vamos criar agora a classe Luta:

|  |  |
| --- | --- |
| **LUTA** | |
| **-** | desafiado |
| **-** | desafiante |
| **-** | rounds |
| **-** | aprovada |
|  | marcarLuta() |
|  | luta() |

**Abaixo temos um exemplo de um objeto composto, ou seja, com um vetor de vários lutadores e seus dados. Primeiro temos a definição dos atributos e métodos do objeto e depois temos o resultado php:**

<?php

class Lutador {

    private $nome;

    private $nacionalidade;

    private $idade;

    private $altura;

    private $peso;

    private $categoria;

    private $vitorias;

    private $derrotas;

    private $empates;

    function \_\_construct($n,$nac,$idad,$alt,$pes,$cat,$vit,$der,$empa){

        $this->nome = $n;

        $this->nacionalidade = $nac;

        $this->idade = $idad;

        $this->altura = $alt;

        $this->peso = $pes;

        $this->categoria = $cat;

        $this->vitorias = $vit;

        $this->derrotas = $der;

        $this->empates = $empa;

    }

    public function apresentar() {

    echo "<p> O lutador se chama " . $this->getNome() . ", sua nacionalidade é " . $this->getNacionalidade() . ", seu peso é <br> " . $this->getPeso() . "sua altura é " . $this->getIdade() . " e possuí vitórias, derrota e empates respectivamente: " . $this->getVitorias() . "," . $this->getDerrotas() . "," . $this->getEmpates() . ". </p>";

    }

    public function ganharLuta(){

    $this->setVitorias($this->getVitorias() + 1);

    }

    public function perderLuta() {

    $this->setDerrotas($this->getDerrotas() + 1);

    }

    public function empatarLuta(){

    $this->setEmpates($this->getEmpates() + 1);

    }

    public function getNome(){

        return $this->nome;

    }

    public function setNome($n){

        $this->nome = $n;

    }

    public function getNacionalidade(){

        return $this->nacionalidade;

    }

    public function setNacionalidade($nac){

        $this->nacionalidade = $nac;

    }

    public function getIdade(){

        return $this->idade;

    }

    public function setIdade($idad){

        $this->idade = $idad;

    }

    public function getAltura(){

        return $this->altura;

    }

    public function setAltura($alt){

        $this->altura = $alt;

    }

    public function getPeso(){

        return $this->peso;

    }

    public function setPeso($pes){

        $this->peso = $pes;

    }

    public function getCategoria(){

        return $this->categoria;

    }

    public function setCategoria($cat){

        $this->categoria = $cat;

    }

    public function getVitorias(){

        return $this->vitorias;

    }

    public function setVitorias($vit){

        $this->vitorias = $vit;

    }

    public function getDerrotas(){

        return $this->derrotas;

    }

    public function setDerrotas($der){

        $this->derrotas = $der;

    }

    public function getEmpates(){

        return $this->vitorias;

    }

    public function setEmpates($empa){

        $this->empates = $empa;

    }

}

**Agora temos o resultado em php na minha página HTML:**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body> <pre>

    <?php

    require\_once 'lutador.php';

    $L = [];

    $L[0] = new Lutador("Wesley","Brasileiro",21,1.75,80.0,"Peso médio",0,0,0);

    $L[1] = new Lutador("Michael","Suíço",20,1.65,60.0,"Peso pena",0,0,0);

    print\_r($L);

    $L[1]->apresentar();

    $L[1]->ganharLuta();

    $L[1]->apresentar();

    ?>

   </pre>

</body>

</html>

* **Relacionamento de Classes PARTE 02:**

Após criarmos a classe lutador com atributos e métodos, encapsulados, agora vamos relacionar este objeto com outro objeto (outra classe), ou seja, relacionamento de classes. Lembre-se, objetos são instâncias de uma classe. Além de encapsular dados e funções, eu consigo relacionar classes. Existe um relacionamento específico que se chama agregação.

Isso se baseia que eu crio uma nova classe, no caso dos lutadores, uma nova classe chamada Luta(). Nela eu crio o atributo “desafiado” e “desafiante”. Dentro desses atributos que criei eu relaciono os atributos dos lutadores da classe anterior Lutador.

Ou seja, eu puxo o atributo de outra classe para esta classe.

Mesmo assim, não esqueça de encapsular (atributos privados e métodos públicos).

Minha nova classe Luta ficaria assim:

|  |  |
| --- | --- |
| **LUTA** | |
| **-** | desafiado |
| **-** | desafiante |
| **-** | rounds |
| **-** | aprovada |
| **+** | marcarLuta() |
| **+** | lutar() |

Estou criando uma relação entre uma classe com uma instância de uma outra classe.

* **Herança em POO:**

Uma herança é a relação entre um elemento pai com um elemento filho. Este elemento filho pode trazer características e comportamentos desses pais.

**Ou seja, a herança permite basear uma nova classe na definição de uma outra classe previamente existente. Eu não preciso criar ela do zero, recriar. Faço ela baseada na primeira classe.**

Exemplo:





**Para ligar esta nova classe pessoa geral com as classes exclusivas usamos a seguinte linha de código em PHP:**

<?php

require\_once 'pessoa.php';

class funcionario extends pessoa {

**Colocamos o “extends” para dizer que nossa classe atual extende-se para a outra classe “pessoa”.**

**Lembre-se: Uma classe abstrata não pode ser instanciada como objeto, ela só serve para ser usada em heranças.**

**Se eu criar o mesmo método ou atributo de uma classe mãe em uma classe filha, a classe filha sobrescreve o que está na classe mãe.**

**Porém se eu colocar “final” na linha do método ou atributo, ele não deixa que seja sobreposto em um elemento filho. Se eu colocar “final” na classe-mãe, ela não poderá ter filho.**

* **Polimorfismo:**

Polimorfismo significa que um mesmo nome defina vários comportamentos diferentes. Várias formas diferentes. Como, nome “acordar” que pode ser de várias maneiras: “irritado”,”feliz”,etc.

Precisamos entender também o que são assinaturas dos métodos. Assinatura dos métodos são a quantidade e os tipos de seus parâmetros.

Exemplos de mesma assinatura:

**Publico método calcMedia(n1:Real, n2:Real): Inteiro;**

**Publico método calcMedia(v1:Real, v2:Real):Real;**

Aqui no exemplo acima eles tem a mesma assinatura (quantidade e tipos de parâmetros iguais), apesar de retornaram tipos diferentes.

Vamos agora aos tipos de polimorfismos:

* **Sobreposição**: Você sobrepor um comando em um elemento filho. Ou seja, acontece quando substituímos um método de uma superclasse na sua subclasse, usando a mesma assinatura; **Mesma assinatura, classes diferentes.**
* **Sobrecarga**: Você sobrepor um valor em um elemento filho.Eu estou sobrecarregando um método. Faço um método de várias formas de assinaturas diferente. **Assinaturas diferentes, mesma classe.**

Ou seja, o polimorfismo é você ter valores e resultados diferentes com um mesmo método.

Assim, abstraindo um pouco a analogia, uma classe é um conjunto de características e comportamentos que definem o conjunto de objetos pertencentes à essa classe. Repare que a classe em si é um conceito abstrato, como um molde, que se torna concreto e palpável através da criação de um objeto. Chamamos essa criação de instanciação da classe, como se estivéssemos usando esse molde (classe) para criar um objeto.