**PHP7 e Orientação a Objetos**

**Considerações preliminares:**

Paradigma – padrão, conduta, uma forma de se fazer alguma coisa;

No PHP podem ser desenvolvidos códigos baseados em um paradigma procedural e um paradigma de orientação a objetos dentro do mesmo script;

Paradigmas esses que são os mais difundidos no mercado.

Podemos distinguir qual é o paradigma empregado nos scripts analisando os princípios aplicados ao código; princípios de cada um:

**-Procedural/Estruturado/Imperativo**

Sequência – interpretação gravitacional e chamada de funções;

Decisão – aplicação de comandos como ifelse ou switch;

Repetição - aplicação de comandos como for, while e do while.

(Funcional e lógico)

**-Orientado a objetos**

Princípios serão falados ao longo da seção;

Princípios são atendidos pelo PHP.

(Concorrente e reativo)

É fato que o PHP suporta o paradigma procedural e de orientação a objetos, pois ele possui os recursos necessários para atender aos princípios dos 2.

**Introdução a orientação a objetos**

***Paradigma Procedural***

Chamada de procedimentos para manipulação de dados (métodos ou funções).

***Paradigma de Orientação a Objetos***

Estrutura de dados com comportamentos (métodos ou funções).

**Princípios -> Pilares:**

**Abstração**: forma como interpretamos as coisas do mundo real e transportamos para dentro da aplicação.

– **Entidade:** compreensão de que um objeto do mundo real deve passar para dentro da aplicação, identificada a partir de um processo de abstração (observação), não é o objeto em si; **Identidade:** Estabelecer uma referência que futuramente nos permitir acessar o objeto; **Características:** atributos que especificam as características da entidade (ex.:tipo, categoria, título, nome...), o nível dos atributos dependem da regra de negócio da aplicação (pode-se ter + ou - atributos de acordo com as necessidades de negócio); **Ações:** métodos/comportamentos da entidade, ações do objeto; se esse objeto fosse um produto (exibirResumoProduto, alterarValorProduto).

//acessando atributos ou métodos (->):

$y->resumirCadFunc ();

**$this-><atributo que quero receber o valor>** é utilizado para recuperar atributo do objeto.

**public $nome = 'José';**

**public $numFilhos = 2;**

**return "$this->nome possui $this->numFilhos filho(s)";**

**//atribuindo ao atributo numFilhos do objeto o valor de $numFilhos que é o parâmetro recebido pelo método(modificarNumFilhos)**

**$this->numFilhos = $numFilhos;**

**class Funcionario**

**{**

**//class - é seguido um padrão onde a primeira letra da palavra é maiúscula, isso segue não importa quantas palavras houver**

**//atributos**

**//primeira palavra sempre minúscula, mas se houver mais de uma, a/s outra/s começam com letra maiúscula**

**public $nome = 'José';**

**public $telefone = '11 99999-8888';**

**public $numFilhos = 2;**

**//métodos - na prática são funções**

**//padrão de definição de nomes segue mesma regra da definição de variáveis-atributos**

**function resumirCadFunc()**

**{**

**}**

**function modificarNumFilhos($numFilhos)**

**{**

**//afetar um atributo do objeto**

**$this->numFilhos = $numFilhos;**

**//atribuindo ao atributo numFilhos do objeto o valor de $numFilhos que é o parâmetro recebido pelo método(modificarNumFilhos)**

**}**

**//variável recebe instância da classe Funcionário**

**$y = new Funcionario();**

**//criando objeto com base no modelo Funcionário e atribuindo ele a variável $y**

**//a partir dessa variável estamos recebendo os atributos e métodos do objeto**

**//acessando atributos ou métodos(->):**

**echo $y->resumirCadFunc(); //exibindo resumo do cadastro do funcionario**

**echo '<br/>';**

**$y->modificarNumFilhos(3); //modificando num de filhos**

**echo $y->resumirCadFunc(); //exibindo  cadastro novamente (com informação de numFilhos atualizada)**

**?>**

**//nova instância de objeto / novo funcionário**

**$x = new Funcionario();**

**echo $x->resumirCadFunc();**

**echo '<br/>';**

**echo $x->modificarNumFilhos(1);**

**echo $x->resumirCadFunc();**

dados distintos dentro da aplicação que podem ser evoluídos de forma isolada.

**É ideal que as classes sejam genéricas, não tenham atributos engessados (atributos com valor fixo)**;

**A ideia seria evoluir os atributos no momento de criação do objeto ou após a criação desse objeto**.

No exemplo desse projeto:

**$nome = null;**

**$telefone = null;**

**$numFilhos = null;**

**Métodos Getters e Setters**

Nos permitem atender a uma convenção que diz qual a melhor forma de acessar e manipular os atributos dos objetos;

Não é uma regra a manipulação de atributos com getter e setter, é opcional;

**Setter** – setar ou **definir** os **valores** de um atributo de um objeto. Recebe e manipula um valor, **sem retorno**. ***Estrutura:***

Palavra reservada **function**;

Nome do método: set<NomeDoMétodo> - sendo que o nome do método **inicia sempre com letra maiúscula** (por sempre começar na segunda palavra);

Abre e fecha parênteses;

Escopo do método definido dentro de chaves.

function setNome(){

}

$this->nome = $nome;

    } //para manipular um atributo do próprio objeto basta utilizar a palavra reservada this e atribui-se a ela o valor passado por parâmetro na função

A palavra this indica que estamos utilizando uma palavra reservada do próprio objeto;

**Getter** – Nos permitem recuperar valores dos atributos;

Mesma **sintaxe do set**, só muda que **ao invés de set é get**;

A palavra utilizada para fazer o retorno da variável é ‘**return’**.

function setNumFilhos($numFilhos)

    {

        $this->numFilhos = $numFilhos;

    }

    function getNome()

    {

        return $this->nome;

    }

$y->setNumFilhos(2);

echo $y->getNome() . ' possui ' . $y->getNumFilhos() . ' filho(s) ';

**Getters e Setters mágicos (overloading de atributos e métodos)**

Nos exemplos dado, na classe funcionário haviam apenas 3 atributos e foram implementados get e set apenas para $nome e $numFilhos. Implementação simples;

Se houvesse uma classe com diversos atributos e para cada um tivéssemos que implementar os métodos getter e setter, fazendo com que fique extenso e a codificação fique redundante;

Pensando nisso, as pessoas começaram a utilizar os métodos getters e setters de uma forma mais inteligente. Utilizando o conceito de overloading ou sobrecarga;

A ideia desse conceito é criar um único método get e um único método set capazes de se adaptar ao atributo que será manipulado.

**Nome de método deve ser essencial para seu fim.** Pois sempre que ele for utilizado dentro da aplicação ou estiver fazendo a manutenção em outras ações, provavelmente o método responsável por fazer o set dos atributos do objeto, terá esse nome.

function \_\_set($atributo, $valor)

    {

    $this->$atributo = $valor;

    }

$y = new Funcionario();

$y->\_\_set('nome', 'Jose');

$y->\_\_set('numFilhos', 2);

echo $y->resumirCadFunc();

echo '<br/>';

echo $y->\_\_get('nome') . ' possui ' . $y->\_\_get('numFilhos') . ' filho(s) ';

**Chamar métodos internamente**

        return $this->\_\_get('nome'). "possui". $this->\_\_get('numFilhos'). "filho(s)";

**Exibindo método (passando variáveis)**

$y = new Funcionario();

$y->\_\_set('nome', 'Jose');

$y->\_\_set('numFilhos', 2);

echo $y->resumirCadFunc();

**Método Construtor e Destrutor (Construct e Destruct)**

Também são conhecidos como métodos mágicos, logo, síntaxe começa com ‘\_\_’;

Esses métodos fazem parte do que consideramos ser o ciclo de vida de um objeto;

Quando a instância de um objeto é criada com base em uma classe, automaticamente o método constructo é executado de modo que uma ação possa ser tomada no processo de inicialização do objeto, sendo essa ação disparada automaticamente em função da instância.

Já o destruct é executado automaticamente quando a instância do objeto é removida da memória, ou seja, quando aquele objeto deixa de existir dentro da aplicação.

***Síntaxe construct***

function \_\_construct()

    {}

***Síntaxe destruct***

function \_\_destruct()

    {

        echo 'Objeto removido';

    }

unset($pessoa); //forçar de forma proposital a remoção desse objeto

método **unset()** e como parâmetro, a **variável que contém** a **referência para** o **objeto instanciado**;

***Em resumo:***

Com os métodos contruct e destruct, podemos criar essa espécie de ‘ciclo de vida’ da aplicação;

O método constructo éa disparado automaticamente na instância do objeto, podendo receber ou não parâmetros;

O método destruct sendo executado automaticamente em função da remoção do objeto da memória. Isso também pode ser feito de forma proposital (intencional) através do método unset, ou esse método pode ser executado pelo simples fato da finalização da interpretação do script.

**Herança:** Compreender oque existe de comum entre os objetos da aplicação que possa ser abstraído e definido em outro modelo;

A ideia é centralizar atributos e métodos comuns para os nossos objetos, de modo a implementar uma outra classe que possa estender esses atributos e métodos para as classes especialistas;

Em alguns casos ouve-se os termos:

- Classe pai e classes filhas;

- Classes genéricas e classes especializadas;

- Super classe e subclasse.

Termos comuns que representam a mesma coisa (classes que **estendem funcionalidades de outra classe**, criando essa **relação de herança** de atributos e métodos).

**- Reutilizável e Manutenção**.

Antes da relação de herança:

class Carro

{

    public $placa = 'ABC1234';

    public $cor = 'Branco';

    public $teto\_solar = true;

    function acelerar()

    {

        echo 'Acelerar';

    }

    function abrirTetoSolar()

    {

        echo 'Abrir teto solar';

    }

    function alterarPosicaoVolante()

    {

        echo 'Alterar posição volante';

    }

}

class Moto

{

    public $placa = 'DEF1122';

    public $cor = 'Preta';

    public $contraPesoGuidao = true;

    function acelerar()

    {

        echo 'Acelerar';

    }

    function empinar()

    {

        echo 'Empinar';

    }

}

Para conectar a classe que contém os atributos e métodos a serem herdados e a classe que quer herdar os mesmos, é só adicionar após o nome desta classe, adicionar:

 extends <classeComAtributoseMetodosDeHeranca>

assim, classe que está sendo criada, herda métodos e atributo da classe que foi colocada após a palavra **extends**;

Depois da relação de herança:

**class Carro extends Veiculo**

**{**

**public $teto\_solar = true;**

**function \_\_construct($placa, $cor)**

**{**

**$this->placa = $placa;**

**$this->cor = $cor;**

**}**

**function abrirTetoSolar()**

**{**

**echo 'Abrir teto solar';**

**}**

**function alterarPosicaoVolante()**

**{**

**echo 'Alterar posição volante';**

**}**

**}**

**class Moto extends Veiculo**

**{**

**public $contraPesoGuidao = true;**

**function \_\_construct($placa, $cor)**

**{**

**$this->placa = $placa;**

**$this->cor = $cor;**

**}**

**function empinar()**

**{**

**echo 'Empinar';**

**}**

**}**

**class Veiculo**

**{**

**public $placa = null;**

**public $cor = null;**

**function acelerar()**

**{**

**echo 'Acelerar';**

**}**

**function freiar()**

**{**

**echo 'Freiar';**

**}**

**}**

Vantagem: reutilização de códigos.

**Polimorfismo**:

Acontece quando trabalhamos com a herança, temos: objeto pai e objeto filho (herda atributos e métodos do objeto pai);

Não necessariamente o objeto filho deve se comportar da mesma forma que o objeto pai, nessas situações, utilizamos o polimorfismo (sobrescrita de métodos) para corrigir os comportamentos do objeto filho.

**- Sobrescrita de métodos.**

Um exemplo seria o caso de dois objetos fazer a mesma coisa, porém de formas diferentes. Função seria passada no objeto pai e utilizaríamos o método de sobrescrita no objeto que faz aquela ação de forma diferente.

class Veiculo

{

function trocarMarcha()

{

echo 'Desengatar embreagem com o pé e engatar marcha com a mão';

}

}

class Moto extends Veiculo

{

function trocarMarcha()

{

echo 'Desengatar embreagem com a mão e engatar marcha com o pé';

}

}

**Encapsulamento**: o objeto poderá o que dentro dele está disponível para o sistema;

Encapsular um objeto, tornando esse objeto **seguro**;

É uma forma de controlar a visibilidade desses atributos e métodos de acordo com a necessidade de negócio;

Podemos **definir se** atributo **é privado ou público**. Criando uma camada de segurança para cada objeto;

Possui dois lados: Conceitual e Técnico;

Quando se tenta acessar um atributo privado ou protegido. Aparece uma mensagem de erro: ***‘não foi possível acessar propriedade privada/protegida do objeto pai’***;

Do ponto do de vista da interação do objeto com a aplicação, private e protected acabam tendo a mesma função. Porém, eles se comportam de forma diferente em um processo de herança;

Para ter acesso e modificação de valores privados ou protegidos na aplicação, basta utilizá-lo dentro de um método público (get e set);

public function setNome($value)

    {

        if (strlen($value >= 3)) {

            $this->nome = $value;

        }

    }

public function setSobrenome($value)

    {

        if (strlen($value >= 3)) {

            $this->nome = $value;

        }

    }

$pai = new Pai();

echo $pai->getNome(); //imprime sobrenome

$pai->setSobrenome('Silva'); //muda sobrenome

echo '<br/>';

echo $pai->getNome(); //imprime sobrenome atualizado

**Segurança.**

Caso haja algo público dentro de uma função privada, o mesmo pode ser exibido.

public function executarAcao()

    {

        $x = rand(1, 10);

        if ($x >= 1 && $x <= 8) {

            $this->responder();

        } else {

            $this->executarMania();

        }

    }

**Encapsulamento – Herança (atributos e métodos são herdados por outro objeto)**

Em um processo de herança atributos e métodos **private não são herdados**;

Em um exemplo em que objeto filho herda atributos do objeto pai, retornamos um array para obter as informações visualmente:

Texto, Carta

Descrição gerada automaticamenteVemos que atributo **nome** (private), **pertence a classe pai**. Logo, se tentarmos recuperar o atributo nome, ocasionará em um erro.

echo $filho->getAtributo('nome'); - erro

$filho->setAtributo('nome', 'Pereira');

echo '<pre>';

print\_r($filho);

echo '</pre>';

echo '<br/>';

echo $filho->getAtributo('nome');

Quando método mágico (\_\_get,\_\_set) definido na classe pai, quando tentamos retornar atributo privado pelo objeto filho utilizando o método mágico, retorna normalmente.

class Pai

{

public function \_\_get($attr)

    {

        return $this->$attr;

    }

    public function \_\_set($attr, $value)

    {

        $this->$attr = $value;

    }

}

Funciona:

echo $filho->\_\_get('nome');

Se métodos \_\_get e \_\_set forem definidos em ‘Filho’, serão utilizados apenas no contexto desse objeto;

Embora o método público tenha sido herdado pelo objeto filho e ao ser executado ele preserve o contexto de um método private do objeto pai, isso não acontece para o operador de visibilidade protected, esse método pode ser sobreposto dentro do objeto filho e automaticamente ter o seu contexto atualizado;

Encapsulamento dentro de herança é muito complexo.

//exibir os métodos do objeto

print\_r(get\_class\_methods())

**Atributos e métodos estáticos**

**public static $atributo1 = 'Eu sou um atributo estático';**

**public $atributo2 = 'Eu sou um atributo normal';**

**public static function metodo1()**

**{**

**echo 'Eu sou um método estático';**

**}**

**public function metodo2()**

**{**

**echo 'Eu sou um atributo normal';**

**}**

**- Não está acessível através do operador *->*;**

**- Não é possível utilizar *$this*;**

Podem ser acessados sem realizar a instância do objeto;

Para definir que o elemento é estático, deve sempre haver **após o operador de visibilidade a palavra static**;

Resgatando método e variável (resolução de escopo):

echo Exemplo::$atributo1;

Exemplo:: metodo1();

Nesse exemplo não é utilizada a função ‘echo’, pois ela já existe dentro da função.

$x = new Exemplo ();

echo $x->atributo1;

No exemplo acima instância funciona, mas não é possível ter acesso ao atributo static.

Essa sintaxe de resolução de escopo pode ser utilizada também em **atributos** normais (não são estáticos) – em métodos normais não é possível utilizá-la;

Em métodos estáticos não é possível utilizar o operador $this (que ajusta contexto das variáveis internas dos métodos para utilizar atributos do objeto);

**Interfaces**

Classes – definem modelo dos objetos;

**Interfaces** – métodos cujo as classes devem obrigatoriamente implementar.

Funcionam como uma espécie de contato, fazendo com que as classes que implementem essas interfaces respeitem os métodos estabelecidos nesses contatos. Fazendo com que a aplicação tenha uma certa homogeneidade.

Dentro da classe que implementa a interface só deve haver métodos (atributos não entram);

Nome **inicia** sempre **com** **caractere maiúsculo**.

***Interface <NomeInterface> {***

***}***

interface EquipamentoEletronicoInterface

{

    public function ligar();

    public function desligar();

}

**Interfaces não implementam os métodos, apenas definem a assinatura daquele método;**

Implementa o método a classe que implementa a respectiva interface;

Todos os **métodos** contidos na interface **precisam ser públicos**.

Implementar mais de uma interface para a mesma classe:

class Humano implements Mamifero, Terrestre

***basta separar por vírgula.***

- Funções não precisam ser declaradas na ordem em que foram declaradas a interface;

- Depois de declaradas as funções das interfaces implementadas, podem ser criadas outros métodos normalmente, inclusive com outras visibilidades (protected, private);

- É possível que interfaces herdem regras estabelecidas em outras interfaces.

Essa herança é feita como a que fazemos com classe.

**interface Animal**

**{**

**public function comer();**

**}**

**interface Ave extends Animal**

**{**

**public function voar();**

**}**

**class Papagaio implements Ave**

**{**

**public function voar()**

**{**

**echo 'Voar';**

**}**

**public function comer()**

**{**

**echo 'Comer';**

**}**

**}**

**Namespaces - 1**

**Não** necessariamente **está relacionado** a **orientação a objetos**;

**Possibilitam agrupamento** de classes, interfaces, funções e constantes;

Visam **evitar** o **conflito entre** seus respectivos **nomes**, utilizar bibliotecas pré-existentes podem resultar em tal conflito.

- No exemplo abaixo, existe uma classe chamada ‘Cliente’ e recebe uma biblioteca que também contém uma classe com esse nome. Esse conflito pode ser resolvido utilizando o **namespace**;

Basta definir que a partir de um certo ponto do script o namespace.

namespace <nome>;

Nesse caso foram definidos dois, um no começo de cada uma das classes;

Isso resolverá o conflito. Porém, não é muito normal ter mais de dois namespace no mesmo script.

**Namespace 2 -> Importando e apelidando namespaces (Use e Aliasing)**

Site – **packagist.org:** está por trás do composer (gerenciador de pacotes do PHP);

Nesse site, é possível encontrar diversos pacotes prontos para lidar com certas situações, como o envio de emails a partir do back-end da nossa aplicação;

Nele, contém biblioteca de envios de email, biblioteca de baixar pdf; tem como baixar frameworks; fazer autenticações com serviços como google, facebook e etc;

Antes de implementar qualquer coisa na aplicação, verificar se não há bibliotecas que atendam a necessidade atual.

**Tratamento de erros**

**Try, Catch, Finally e Throw**

Serve para tornar a aplicação mais inteligente na ocorrência de erros, fazendo com que a experiência final do usuário seja mais agradável;

É importante entender bem a técnica e tratamento de erros, pois com elas pode-se ter aplicações cada vez mais seguras.

try (tente) -> erro -> catch (pegar)

finally – finalmente,

throw – lançar.

try – encapsula todo o conteúdo suscetível a algum erro onde nós podemos identificar e controlar uma exceção.

Dentro de um mesmo script podem ser utilizados diversos try’s em momentos diferentes;

//uma lógica

try {

//uma lógica onde possa ocorrer um potencial erro(exceção)

}

//uma lógica

try {

//uma lógica onde possa ocorrer um potencial erro(exceção)

}

O try pode ter dois destinos:

**catch** – captura um eventual erro ocorrido dentro do bloco try;

**finally** – instrução final do tratamento de erros.

**Catch** **após** ser **finalizado** também **vai para finally**;

No exemplo abaixo:

Try – recuperando erro;

Catch – adicionando alguma tratativa para esse erro;

Finally – dando continuidade à tratativa do script.

**try {**

**echo ' <h3> \*\*\*Try\*\*\* </h3> ';**

**$sql = 'Select \* from clientes';**

**mysql\_query($sql); //erro**

**//uma lógica onde possa ocorrer um potencial erro(exceção)**

**} catch (Error $e) {**

**echo ' <h3> \*\*\*Catch\*\*\* </h3> ';**

**echo '<p style=color:red>' . $e . '</p>';**

**} finally {**

**echo ' <h3> \*\*\*Finally\*\*\* </h3> ';**

**}**

Try não pode estar vazio, se estiver pode haver um fatal error;

Dentro do catch basicamente estamos **trabalhando** o **erro**, **melhorando** a **resposta**, o **feedback para** o **usuário final** e **armazenando** esse **erro no banco de dados**;

Armazenando esses erros em banco de dados, ele pode ser analisado pela equipe de desenvolvimento, de modo a corrigir potenciais fontes de bug no código;

**Tratamento de erros é essencial na aplicação**;

Quando se utiliza **try** e **catch**, **finally** é **opcional**.

**Throw** – possibilidade de lançar erros intencionalmente.

if(!file\_exists('require\_arquivo\_a.php')){

        throw new Exception();

    }//Exception - objeto // a classe desse objeto já está definida dentro do PHP

new – operador de instância.

Exception – programador em questão lança;

Error – nativo do PHP.

**Tratamento de erros – exceções customizadas**

class MinhaExceptionCustomizada extends Exception

//como Excepion é uma classe nativa do PHP, logo, podemos criar classes que extendem ela.

Exception pode ser estendida por outra classe.

    throw new MinhaExceptionCustomizada('Esse é um erro de teste');

Criando objeto e passando parâmetro para o construtor desse objeto

private $erro = '';

    public function \_\_construct($erro)

    {

        $this->erro = $erro;

    } //ajusta contexto para recuperar atributo 'erro' do objeto // atribui a ele o valor recebido por parâmetro no construtor

Ajusta contexto para receber valor do objeto em questão, recebe valor do método criado e atribui esse parâmetro ao atributo *erro* do objeto.

public function exibirMensagemErroCustomizada()

    {

        return $this->erro;

    }

Retorna atributo erro do objeto, para que seja possível exibir mensagem na variável $e

catch (MinhaExceptionCustomizada $e) {

    echo $e->exibirMensagemErroCustomizada();

}

Exibindo mensagem de erro

Não é possível recuperar atributo *erro* diretamente, pois ele é privado.

public function exibirMensagemErroCustomizada()

    {

        echo '<div style="border: 1px solid #000; padding: 15px; background-color:red; color:write;">';

        echo $this->erro;

        echo "</div>";

customizando erro

**Resumo:**

**Error (php) –** base para lançamentos de erros internos do PHP -> funções nativas do PHP que tratam erros e podem eventualmente lançar erros na execução, implementam objetos de erro baseado na classe **Error**.

**Exception (programadores)** – Geralmente serão lançadas **exceções** para serem **capturadas** **no catch**.

**Customizadas (programadores)** – Também desenvolvida pelo programador para lançamento de erros mais personalizados. É comum que bibliotecas prontas implementem classes de erros customizadas (ficar atento se tipagem está sendo feita de acordo com a classe customizada).

**App Send Mail**

Paradigma de **O**rientação a **O**bjetos;

Recursos, como: **namespace**, **tratamento de erros**;

**Incorporar biblioteca** PHPMaler;

**Segurança** do lado do back-end.

Ao término, teremos uma aplicação funcional capaz de **enviar Email através do serviço de SMTP do Gmail**.

Formulário:

**Para**: <nome de um destinatário **válido**>;

**Assunto**;

**Mensagem**;

**Enviar**.