Introdução ao PHP Orientado a Objetos

Índice

Advertência

Esta apostila é fruto de pesquisa por vários sites e autores e com alguma contribuição pessoal. Geralmente o devido crédito é concedido, exceto quando eu não mais encontro o site de origem. Isso indica que esta apostila foi criada, na prática, "a várias mãos" e não somente por mim. Caso identifique algo que seja de sua autoria e não tenha o devido crédito, poderá entrar em contato comigo para dar o crédito ou para remover: ribafs @ gmail.com (remova os dois espaços).

É importante ressaltar que esta apostila é distribuída gratuitamente no repositório:

https://github.com/ribafs/apostilas

Sob a licença MIT. Fique livre para usar e distribuir para qualquer finalidade, desde que mantendo o crédito ao autor.

Sugestões

Sugestões serão bem vindas, assim como aviso de erros no português ou no PHP. Crie um issue no repositório ou me envie um e-mail ribafs @ gmail.com.

Como ser um bom programador

É fácil seguir uma apostila como esta e criar aplicativos, pois ela foi testada e provavelmente não aparecerá erros. Mas não tem jeito, eles sempre aparecem. As mensagens de erro geralmente ajudam mas nem sempre são precisas. Neste momento precisamos ter um espírito tipo detetive e sair rastreando o problema pelas trilhas mais prováveis até encontrar o erro e corrigir.

É nestes momentos que a programação mostra para alguns que eles estão no lugar errado e que devem fazer outra coisa da vida.

Caso realmente queira ser programador, empenhe-se, organize-se e siga seus instintos de detetive.

Costumo dizer que pegar um exemplo e executar é fácil, mas que nem sempre é fácil corrigir erros. Mas um programador não desiste com dificuldades, elas o estimulam a corrigi-las, tornando-se assim melhores.

Autor

Ribamar FS

ribafs @ gmail.com

https://ribamar.net.br

https://github.com/ribafs

Fortaleza, 17 de dezembro de 2021

Recomendações

O conhecimento teórico é importante para que entendamos como as coisas funcionam e sua origem, mas para consolidar um conhecimento e nos dar segurança precisamos de prática e muita prática.

Não vale muito a penas ser apenas mais um programador. É importante e útil aprender muito, muito mesmo, ao ponto de sentir forte segurança do que sabe e começar a transmitir isso para os demais.

Tenha seu ambiente de programação em seu desktop para testar com praticidade todo o código desejado.

Caso esteja iniciando com PHPOO recomendo que leia por inteiro e em sequência. Mas se já entende algo dê uma olhada no índice e vá aos assuntos que talvez ainda não domine.

Não esqueça de ver e testar também o conteúdo da pasta Anexos do repositório.

Caso tenha alguma dúvida, me parece que a forma mais prática de receber resposta é através de grupos. Depois temos o Google.

Dica: quando tiver uma dúvida não corra para pedir ajuda. Antes analise o problema, empenhe-se em entender o contexto e procure você mesmo resolver. Assim você está passando a responsabilidade para si mesmo, para seu cérebro, que passará a ser mais confiante e poderá te ajudar ainda mais. É muito importante que você confie em si mesmo, de que é capaz de solucionar os problemas. Isso vai te ajudar. Somente depois de tentar bastante e não conseguir, então procure ajuda.

Veja que este material não tem vídeos nem mesmo imagens. Isso em si nos nossos dias é algo que atrai pouca gente, pois vídeos e fotos são mais confortáveis de ler e acompanhar. Ler um texto como este requer mais motivação e empenho. Lembrando que para ser um bom programador precisamos ser daqueles capaz de se empenhar e suportar a leitura e escrita também.

Autodidata

Não tive a pretensão de que esta apostila fosse completa, contendo tudo sobre PDO, que seria praticamente impossível. Aquele programador que quando não sabe algo seja capaz de pesquisar, estudar, testar e resolver praticamente sozinho. Este é um profissional importante para as empresas e procurado.

1 – Introdução

A programação orientada a objetos não é uma linguagem mas sim um paradigma adotado por várias linguagens, uma forma de programar. Tanto que PHP orientado a objetos tem a maior parte do PHP estruturado.

Como a maioria das atividades que fazemos no dia a dia, programar também possui modos diferentes de se fazer. Esses modos são chamados de paradigmas de programação e, entre eles, estão a programação orientada a objetos (POO) e a programação estruturada. Quando começamos a utilizar linguagens como PHP, Java, C#, Python e outras que possibilitam o paradigma orientado a objetos, é comum errarmos e aplicarmos a programação estruturada achando que estamos usando recursos da orientação a objetos.
https://www.alura.com.br/artigos/poo-programacao-orientada-a-objetos

Para iniciar esta introdução, vejamos um pouco de história: Como o PHP não é uma linguagem que foi criada para ser orientada a objetos (só começou a suportar orientação a objetos na versão 3, sendo aprimorada na versão 4, na versão 5.3e o suporte a orientação a objetos está excelente), os programadores PHP utilizavam ou a programação estruturada ou orientação a funções (nomenclatura usada por estudantes para definir um método de desenvolvimento). Este método basicamente organiza as funções mais utilizadas em arquivos específicos, como por exemplo, um arquivo chamado funções de banco e neste arquivo são colocadas as funções de insert, update e delete, depois bastava incluir o arquivo no local onde deseja utilizar as funções. Para isso utiliza-se os métodos include, include_once, require ou require_once do PHP e chamar as funções.

O include tenta incluir o arquivo, caso não ache, retorna um Warning (warning é apenas um alerta do PHP, a aplicação não é interrompida quando acontece). O require por sua vez retorna um Fatal Error (o fatal error interrompe a aplicação e não executa o resto dos comandos), o include_once e require_once tentam incluir o arquivo, porém se o arquivo já foi incluso ele retorna false e não o inclui novamente.

Saindo um pouco da parte histórica e indo para a parte mais acadêmica, vamos estudar um pouco dos conceitos básicos de orientação a objetos para em seguida fazer alguns trechos de código para fixar melhor os conceitos. O primeiro e mais importante conceito de orientação a objetos é a classe, uma abstração do software de objetos similares, ou seja, um template do qual os objetos serão criados. Crie um arquivo chamado usuário.php para começarmos os exemplos abaixo.
https://www.devmedia.com.br/introducao-a-orientacao-a-objetos-em-php/26762

A orientação a objetos (OO) é um padrão de programação em que um software não é composto por um grande bloco de códigos específicos, e sim de vários blocos de códigos distantes e independentes, que juntos montam um sistema. O PHP faz parte de um grupo de linguagens que possuem suporte a OO, mas não é preso a ela.

Esse padrão não possui um objetivo único e tem como objetivo:

Reutilização de código (tempo economizado);
Escalabilidade (código adicionado mais facilmente);
Manutenibilidade (manutenção mais fácil);
Desenvolvimento mais rápido.

Orientação a objetos e PDO no PHP

A orientação a objetos (OO) é um padrão de programação em que um software não é composto por um grande bloco de códigos específicos, e sim de vários blocos de códigos distantes e independentes, que juntos montam um sistema. O PHP faz parte de um grupo de linguagens que possuem suporte a OO, mas não é preso a ela.

Esse padrão não possui um objetivo único e tem como objetivo:

Reutilização de código (tempo economizado);
Escalabilidade (código adicionado mais facilmente);
Manutenibilidade (manutenção mais fácil);
Desenvolvimento mais rápido.

A POO possui alguns conceitos fundamentais para seu desenvolvimento:

Abstração: são tipos abstratos de dados;
Objetos: um objeto é uma entidade do mundo real, concreta ou abstrata, que seja aplicável ao sistema;
Classes: uma classe é a representação de vários objetos que possuem as mesmas características e comportamentos;
Atributos / Propriedades: são as características de um determinado objeto.

Classes e Objetos

Algo que confunde bastante novos estudantes de OO é a diferença entre Classes e Objetos.

As classes definem as características e o comportamento dos seus objetos. Cada característica é representada por um atributo e cada comportamento é definido por um método.

Então uma classe não é um objeto e sim uma abstração de sua estrutura, no qual podemos definir quantos objetos desejamos ter.

Para podermos entender melhor o funcionamento, vamos criar a nossa primeira classe e alguns objetos a partir dela.
https://www.devmedia.com.br/orientacao-a-objetos-e-pdo-no-php/32644

Orientação a Objetos

Na orientação a objetos temos conceitos essenciais como Classe, Objetos/Instâncias, Atributos, Métodos, Herança, Encapsulamento, Associação, Composição, Agregação, Abstração, Polimorfismo e Interface. Esses são alguns dos conceitos essenciais, nesse primeiro artigo de POO. Porém, não falaremos de todos.

Para muitos programadores PHP, orientação a objetos é um conceito amedrontador, cheio de sintaxes complicadas e pontos de paradas. Portanto para se livrar disso é importante conhecer os conceitos ligados à orientação a objetos e aprender a utilizar, assim entendendo o paradigme.

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. Um dos maiores problemas para os desenvolvedores é ter de manter códigos onde os dados são declarados e redeclarados, acabando num jogo de pique esconde, em busca de funcionalidades e dados duplicados pelo código.

A orientação à objetos é uma maneira de programar que modela os processos de programação de uma maneira próxima à realidade, tratando a cada componente de um programa como um objeto, com suas características e funcionalidades. O tratamento de objetos no PHP 5 foi totalmente reescrito, permitindo a utilização de uma maior quantidade de recursos da POO, melhor performance e mais vantagens na implementação deste tipo de programação.

A POO também possibilita uma maior otimização e reutilização do código.

Programação Orientada à Objetos (POO) com PHP Diretoria de Transferência Tecnológica – Centro de Computação – Unicamp

Classes e Objetos

Classe é a estrutura mais fundamental para a criação de um objeto. Uma classe nada mais é do que um conjunto de variáveis (propriedades ou atributos) e funções (métodos), que definem o estado e o comportamento dos objetos. Quando criamos uma classe, temos como objetivo final a criação de objetos, que nada mais são do que representações dessa classe em uma variável do tipo objeto.

A programação orientada a objetos é um estilo de programação no qual é comum agrupar todas as variáveis e funções de um determinado tópico em uma única classe, ou seja, cada classe sobre um único assunto. A programação orientada a objetos é considerada mais avançada e eficiente do que o estilo procedural de programação. Essa eficiência decorre do fato de que oferece suporte a uma melhor organização de código, fornece modularidade e reduz a necessidade de nos repetir. Dito isso, podemos ainda preferir o estilo procedural em projetos pequenos e simples. No entanto, à medida que nossos projetos crescem em complexidade, é melhor usar o estilo orientado a objetos.

Como criar classes?
Para criar uma classe, agrupamos o código que trata de um determinado tópico em um único lugar. Por exemplo, podemos agrupar todo o código que trata os usuários de um blog em uma classe, todo o código que está envolvido com a publicação das postagens no blog em uma segunda classe e todo o código que é dedicado a comentários em uma terceira classe.

Para nomear a classe, é comum usar um substantivo no singular que começa com uma letra maiúscula. Por exemplo, podemos agrupar um código que lida com usuários em uma classe User, o código que lida com postagens em uma classe Post e o código que é dedicado a comentários em uma classe Comment.

O que são OOPs?
Orientado a Objetos é uma abordagem de desenvolvimento de software que modela a aplicação em torno de objetos do mundo real, como funcionários, carros, contas bancárias, etc. Uma classe define as propriedades e métodos de um objeto do mundo real. Um objeto é uma ocorrência de uma classe.

Os três componentes básicos da orientação a objetos são;
- Análise orientada a objetos - funcionalidade do sistema
- Projeto orientado a objetos - arquitetura do sistema
- Programação orientada a objetos - implementação do aplicativo

Princípios de programação orientada a objetos

Os três princípios principais de OOP são;
- Encapsulamento - trata de ocultar os detalhes de implementação e apenas expor os métodos. O principal objetivo do encapsulamento é;

◦ Reduza a complexidade do desenvolvimento de software - ocultando os detalhes de implementação e apenas expondo as operações, o uso de uma classe torna-se mais fácil.

◦ Proteja o estado interno de um objeto - o acesso às variáveis da classe é feito por meio de métodos como get e set, o que torna a classe flexível e fácil de manter.

◦ A implementação interna da classe pode ser alterada sem a preocupação de quebrar o código que usa a classe.

• Herança - diz respeito ao relacionamento entre as classes. O relacionamento assume a forma de pai e filho. O filho usa os métodos definidos na classe pai. O principal objetivo da herança é;

◦ Reutilização– vários filhos podem herdar do mesmo pai. Isso é muito útil quando temos que fornecer uma funcionalidade comum, como adicionar, atualizar e excluir dados do banco de dados.

• Polimorfismo - trata de ter uma única forma, mas muitas maneiras de implementação diferentes. O principal objetivo do polimorfismo é;

◦ Simplifique a manutenção de aplicativos e torne-os mais extensíveis.

Conceitos OOPs em PHP

PHP é uma linguagem de script orientada a objetos; ele apóia todos os princípios acima. Os princípios acima são alcançados via;

• Encapsulamento - por meio do uso dos métodos “get” e “set”, etc.
• Herança - por meio do uso da palavra-chave extends
• Polimorfismo - por meio do uso da palavra-chave implements

Agora que temos o conhecimento básico de OOP e como ele é suportado no PHP, vamos dar uma olhada em exemplos que implementam os princípios acima

https://www.guru99.com/object-oriented-programming.html

Classes e objetos PHP

Neste tutorial, você aprenderá como escrever código no estilo orientado a objetos em PHP.

O que é programação orientada a objetos

A Programação Orientada a Objetos (OOP) é um modelo de programação baseado no conceito de classes e objetos. Ao contrário da programação procedural, onde o foco está em escrever procedimentos ou funções que realizam operações nos dados, na programação orientada a objetos o foco está nas criações de objetos que contêm dados e funções juntos.

A programação orientada a objetos tem várias vantagens sobre o estilo de programação convencional ou procedural. Os mais importantes estão listados abaixo:

• Fornece uma estrutura modular clara para os programas.
• Ajuda você a aderir ao princípio "não se repita" (DRY) e, assim, tornar seu código muito mais fácil de manter, modificar e depurar.
• Torna possível criar um comportamento mais complicado com menos código e menor tempo de desenvolvimento e alto grau de reutilização.

As seções a seguir descreverão como classes e objetos funcionam em PHP.

Dica: A ideia por trás do princípio Don't Repeat Yourself (DRY) é reduzir a repetição de código abstraindo o código que é comum para o aplicativo e colocando-o em um único lugar e reutilizando-o em vez de repeti-lo.

Compreendendo classes e objetos

Classes e objetos são os dois aspectos principais da programação orientada a objetos. Uma classe é uma coleção autocontida e independente de variáveis e funções que trabalham juntas para realizar uma ou mais tarefas específicas, enquanto os objetos são instâncias individuais de uma classe.

Uma classe atua como um modelo ou projeto a partir do qual muitos objetos individuais podem ser criados. Quando objetos individuais são criados, eles herdam as mesmas propriedades e comportamentos genéricos, embora cada objeto possa ter valores diferentes para certas propriedades.

Por exemplo, pense em uma classe como a planta de uma casa. A planta em si não é uma casa, mas é uma planta detalhada da casa. Enquanto, um objeto é como uma casa real construída de acordo com esse projeto. Podemos construir várias casas idênticas a partir da mesma planta, mas cada casa pode ter diferentes tintas, interiores e famílias em seu interior, como mostra a ilustração abaixo.

Para declararmos uma classe, utilizamos a palavra-chave class.

https://www.uniaogeek.com.br/poo-no-php-parte-1/

2 – Conceitos

Classe - modelo de onde se derivam os objetos. Como uma planta de casa através da qual se constrói casas.

Objetos - derivados das classes. Cada vez que instanciamos uma classe criamos um objeto da mesma. Não se usa classes na programação, mas somente objetos.

Propriedades - são correspondentes às variáveis na linguagem estruturada

Métodos - são similares as funções da estruturada

Visibilidade de propriedades e métodos:
- private - visível somente dentro da classe que a criou e definiu
- protected - visível somente na classe que o criou ou nas classes que a estendem
- public - visível em qualquer objeto que o use (esta é a visibilidade default. Caso não usemos uma explicitamente a public será usada)

Construtor

É um método que utiliza o nome reservado __construct() e que não precisa ser chamado da forma convencional, pois é executado automaticamente quando instanciamos um objeto a partir de uma classe. Sua utilização é indicada para rotinas de inicialização. O método construtor se encarrega de executar as ações de inicialização dos objetos, como por exemplo, atribuir valores a suas propriedades.

O método construtor da classe Noticia é herdado e executado automaticamente na subclasse NoticiaPrincipal. Porém, as "características específicas de NoticiaPrincipal não serão inicializadas pelo método construtor da classe pai". Outro detalhe importante: caso a subclasse NoticiaPrincipal tenha declarado um método construtor em sua estrutura, este mesmo método da classe Noticia não será herdado. Nesse caso podemos chamar o método construtor da classe Noticia, através de uma chamada específica:
parent::__construct()

```
public function __construct(){
   parent::__construct();
}
```

Destrutor

O método destrutor será chamado assim que todas as referências a um objeto em particular forem removidas ou quando o objeto for explicitamente destruído. O método __destruct é executado toda vez que um objeto da classe é destruído pelo fim do script ou através da função unset. Sua função é basicamente zerar variáveis, limpar dados de sessões, fechar conexões com banco de dados, etc.

Como no método construtor, o método destrutor possui um nome reservado, o

__destruct()

Herança

A herança representa uma das principais características da Orientação a Objetos, até porque, somos capazes de implementar tipos de dados hierarquicamente. Através do conceito de herança, conseguimos implementar classes de uso geral, que possuam características comuns a várias entidades relacionadas.

Essas classes poderão ser estendidas por outras, produzindo assim, classes mais especializadas e, que implementem funcionalidades que as tornam únicas.

Através da herança, poderemos utilizar propriedades e métodos definidos na superclasse. Uma boa maneira de pensarmos neste conceito é sob a perspectiva de obter objetos mais especializados conforme aumente a hierarquia. Devemos tomar cuidado com o conceito de hereditariedade animal, até porque, os filhotes não possuem, necessariamente, as características dos pais. Já, o conceito de herança na Orientação a Objetos define que, todo herdeiro receberá o conjunto de características definidas como público e privado e, terá acesso total as funcionalidades definidas na superclasse. Assim, a única maneira de restringir os herdeiros é definindo membros privados, até porque, do contrário, todo e qualquer herdeiro poderá alterar quaisquer informação.

É comum que classes derivadas sejam novamente utilizadas como base para outras. Assim, somos capazes de estender qualquer classe que não tenha o seu construtor definido como privado.

Se tomarmos como exemplo a ideia de frutas, temos que a classe fruta conterá o código que define as propriedades e funções de todas as frutas, enquanto que a classe Maçã, receberá as funções e atributos de todas as frutas, e implementará as propriedades e funções que somente as maçãs possuem.

Toda classe poderá ser herdada e para isso, não é preciso fazer nada de especial, ou seja, o uso da herança se resume a definição explicita na declaração de uma nova classe que a mesma será uma "continuação" de outra.

Classes que são herdadas são chamadas de Classe Base, Super, SuperClasse. Classe herdeira são chamadas de Classes Derivadas, ou SubClasse. Também é comum chamarmos as classes base de superclasses, e as classes que herdam desta, como sendo subclasses.

O PHP não suporta herança múltipla, porém, o mesmo disponibiliza a utilização de traits, que permite a definição de conjuntos de características. Traits serão estudadas posteriormente e também, não podemos confundir o conceito de herança múltipla com traits, até porque, numa análise superficial há semelhanças, mas em teoria, são conceitos distintos.

A sintaxe para uso da Herança

Uma classe herdeira, deverá, por definição, utilizar a instrução extends e, em seguida, definir a classe que será estendida. O código a seguir faz uma breve demonstração da nomenclatura que deve ser utilizada.
https://excript.com/php/heranca-php.html

Namespaces

A comunidade PHP tem muitas pessoas desenvolvedoras criando muito código. Isso significa que o código de uma biblioteca PHP pode usar o mesmo nome de classe que uma outra biblioteca. Quando ambas bibliotecas são usadas no mesmo namespace, elas

colidem e causam problemas.

Os Namespaces resolvem esse problema. Como descrito no manual de referência do PHP, os namespaces podem ser comparados com os diretórios dos sistemas operacionais, que fazem namespace dos arquivos; dois arquivos com o mesmo nome podem coexistir em diretórios separados. Da mesma forma, duas classes PHP com o mesmo nome podem coexistir em namespaces PHP separados. Simples assim.
É importante que você use namespace no seu código para que ele possa ser usado por outras pessoas sem risco de colisão com outras bibliotecas.

Um modo recomendado de usar namespaces está descrito na PSR-4, que tem como objetivo fornecer uma convenção padrão para arquivos, classes e namespaces para permitir um código plug-and-play.

Em outubro de 2014 o PHP-FIG (Framework Interop Group) descontinuou o padrão para autoloading anterior: a PSR-0. Tanto a PSR-0 e a PSR-4 ainda são perfeitamente utilizáveis. A última requer o PHP 5.3, então muitos projetos que rodam apenas em PHP 5.2 implementam a PSR-0. Se você estiver planejando usar um padrão para auto-carregamento para uma nova aplicação ou pacote, olhe na PSR-4

Getters e Setters

Idealmente todas as propriedades devem ser private. Então para passar o valor das propriedades para os objetos precisamos usar métodos public, chamados getter e setter, que são métodos public, com a seguinte convenção para nomes:
getNomePropriedade()
setNomePropriedade()

O getter devolve o valor de uma propriedade.
O setter recebe um novo valor para uma propriedade e altera o valor existente para este.

Encapsulamento

Este recurso possibilita ao programador restringir ou liberar o acesso às propriedades e métodos das classes. A utilização deste recurso só se tornou possível à partir do PHP 5. Aplica-se este conceito através dos operadores:

Public : Quando definimos uma propriedade ou método como public, estamos dizendo que suas informações podem ser acessadas diretamente por qualquer script, a qualquer momento. Até este momento, todas as propriedades e métodos das classes que vimos foram definidas dessa forma.

Protected : Quando definimos em uma classe uma propriedade ou método do tipo protected, estamos definindo que ele só poderão ser acessados pela própria classe ou por classes herdeiras, sendo impossível realizar o acesso externo por um objeto de qualquer classe, por exemplo.

Private : Quando definimos propriedades e métodos do tipo private, só a própria classe pode realizar o acesso, sendo ambos invisíveis para herdeiros ou para classes e programas externos.

Este é um conceito importante e que ajuda na segurança da aplicação.
Basicamente, podemos dizer que esta técnica permite esconder determinadas propriedades do nosso objeto que não deveriam ser visíveis para outras partes do sistema que não deveriam ter acesso a elas.

Métodos e propriedades estáticas

Quando definimos métodos ou propriedades como estáticos (utilizando a palavra-chave static), estamos permitindo que estes possam ser chamados externamente sem haver a necessidade de estarem no contexto de um objeto, isto é, não é necessário instanciar um objeto para poder acessá-los. Para ter acesso a uma propriedade estática dentro do corpo da classe temos que usar a palavra self acompanhada de :: (self::).

Overloading/Sobrecarga

De métodos e de propriedades

Sobrecarga em PHP provê recursos para "criar" dinamicamente membros e métodos. Estas entidades dinâmicas são processadas via métodos mágicos que podem estabelecer em uma classe para vários tipos de ações.

Os métodos sobrecarregados são invocados quando interagem com membros ou métodos que não foram declarados ou não são visíveis no escopo corrente. O resto desta seção usará os termos "membros inacessíveis" e "métodos inacessíveis" para se referirir a esta combinação de declaração e visibilidade.

Todos os métodos sobrecarregados devem ser definidos como públicos.

__set() é executado ao se escrever dados para membros inacessíveis.

__get() é utilizados para ler dados de membros inacessíveis.

Herança – Method Overriding

Definição de Method Overriding:
É quando a função da classe base é redefinida com o mesmo nome, assinatura e especificador de acesso (public ou protected) da classe derivada.

A razão de fazer isso é para prover funcionalidades adicionais sobre as definidas na classe base.

Exemplo prático: você tem uma classe com nome Bird da qual derivam duas outras classes: Eagle e Swift. A classe Bird tem definidos os métodos defined para eat, fly, etc, mas cada uma desstas classes será especializada para Eagle e Swift e deve ter seu próprio estilo de voar que você precisa override as funcionalidades de voar.

Sobrescrevendo Métodos e Propriedades Herdadas

Para alterar uma propriedade ou o comportamento de um método existente na nova classe, você pode, simplesmente, sobrescreve-los, bastando redeclará-los na nova classe:

Polimorfismo em PHP

Neste tutorial, vamos aprender sobre Polimorfismo (do grego para "muitas formas"), uma convenção de nomenclatura que pode nos ajudar a escrever um código muito mais coerente e fácil de usar. De acordo com o princípio do polimorfismo, métodos em classes diferentes que fazem coisas semelhantes devem ter o mesmo nome.

De acordo com o princípio do polimorfismo, métodos em classes diferentes que fazem coisas semelhantes devem ter o mesmo nome.

Um bom exemplo são as classes que representam formas geométricas (como retângulos, círculos e octógonos) que são diferentes entre si no número de costelas e na fórmula que calcula sua área, mas todas têm em comum uma área que pode ser calculado por um método. O princípio do polimorfismo diz que, neste caso, todos os métodos que calculam a área (e não importa para qual forma ou classe) teriam o mesmo nome.

Por exemplo, podemos chamar o método que calcula a área calcArea () e decidir que colocamos, em cada classe que representa uma forma, um método com este nome que calcula a área de acordo com a forma. Agora, sempre que quisermos calcular a área para as diferentes formas, chamaremos um método com o nome de calcArea () sem ter que prestar muita atenção aos detalhes técnicos de como realmente calcular a área para as diferentes formas. A única coisa que precisaríamos saber é o nome do método que calcula a área.

Como implementar o princípio do polimorfismo?

Para implementar o princípio do polimorfismo, podemos escolher entre classes abstratas e interfaces.

Para garantir que as classes implementem o princípio do polimorfismo, podemos escolher entre uma das duas opções de classes abstratas ou interfaces.

No exemplo dado abaixo, a interface com o nome de Shape confirma todas as classes que a implementam para definir um método abstrato com o nome de calcArea ().

Importante em termos de segurança das informações:
- Use somente private
- Caso não consiga atender, então use protected
- Somente se não atender use public

Toda definição de classe
- começa com a palavra-chave class,
- seguido por um nome da classe, que pode ser qualquer nome que não seja uma palavra reservada no PHP,
- seguido por um par de chaves, que contém a definição dos membros e métodos da classe.

$this - Uma pseudo variável, $this, está disponível quando um método é chamado dentro de um contexto de objeto. $this é uma referência para o objeto chamador do método (normalmente o objeto ao qual o método pertence, mas pode ser outro objeto, se o método é chamado estaticamente no contexto de um objeto secundário).

Classe Abstrata
Classe que serve de base para outras classe. Classe que não pode ser instanciada.
Toda classe que incorpora algum método abstrato deve ser declarada como abstrata.
Uma classe que herde de uma classe abastrata deve definir os métodos abstratos declarados na classe abstrata. Do contrário, a classe que herda seria obrigada a ser declarada como abstrata.
Quando uma classe herda de uma classe abstrata, todos os métodos marcados como abstratos na declaração da classe pai devem ser definidos na classe filha e esses métodos devem ser definidos com a mesma visibilidade ou mais pública. Se private então private, protected ou public. Se protected protected ou public. Se public então public.

Uma classe abstrata é um modelo, para que outras classes possam herdá-las.
Uma classe abstrata pode fornecer métodos abstratos e comuns.

Método Abstrato
Método abstrato, é um método que fornece assinatura para um método, mas sem nenhuma implementação.

Qualquer classe que possuir métodos abstrados também deve ser abstrata.
Da mesma forma que acontece com as classes, os métodos abstratos são criados apenas para ajudarem a estruturar as classes filhas.

A implementação deverá ser feita nas subclasses.
O principal uso de classes e métodos abstratos é garantir que cada subclasse sobrescreva os métodos abstratos da superclasse.

Os métodos comuns, que são totalmente implementados em uma classe abstrata, podem ser usados por uma classe concreta por meio de herança.

Classes Final
Como o nome sugere, uma classe final é uma classe que não pode ser instanciada.

```
final class NomeClasse
{

}
```

Métodos final
Quando um método é dito como final ele não pode ser sobrescrito ou seja, a classe filha continua tento acesso a ele, mas não pode alterá-lo.

Interfaces
Uma interface é como um contrato que as classes que a implementam devem cumprir.

Um método na interface não pode conter conteúdo.
Caso haja variáveis, estas devem existir na interface e na classe que a implementa.
A visibilidade do método da interface tem que ser mais restrito ou igual a da classe que a implementa.

Constantes numa interface não podem ser mudadas na classe que a implementa.
É possível uma classe implementar mais de 1 interface ao mesmo tempo , sendo assim , a classe tem que estar nos padrões de todas as interfaces que ela implementa.

Uma interface pode herdar de várias outras Interfaces, sendo assim , a classe que implementa esta interface tem que estar nos padrões de todas as interfaces herdadas também.

No PHP, as interfaces são nada mais do que um modelo de assinaturas de método para outras interfaces ou classes concretas implementarem. Uma interface é implementada usando o operador implements.

Se uma classe implementa uma interface ela deve implementar completamente todos os métodos da interface.

3 - Classes, Objetos, Propriedades, Métodos e Constantes

3.1 - Classe

Uma classe é como um modelo de onde criamos objetos. Sempre que instanciamos uma classe criamos um objeto da mesma.

Toda definição de classe
- começa com a palavra-chave class,
- seguido por um nome da classe, que pode ser qualquer nome que não seja uma palavra reservada no PHP, - seguido por um par de chaves, que contém a definição dos membros e métodos da classe.

$this
Uma pseudo variável, $this, está disponível quando um método é chamado dentro de um contexto de objeto. $this é uma referência para o objeto chamador do método (normalmente o objeto ao qual o método pertence, mas pode ser outro objeto, se o método é chamado estaticamente no contexto de um objeto secundário).

```
<?php
class Pessoa
{

}
?>
```

3.2 - Propriedades

As variáveis de uma classe são chamadas de propriedades. Também chamadas de atributos ou campos.

```
class Pessoa {
	// Uma propriedade (variável)
	private $nome;
}
```

3.3 - Método

Um método na orientação a objetos é similar a uma função na linguagem procedural, tanto que é criado com a palavra function. Veja o método acordar()

```
class Pessoa {
	// Uma propriedade (variável)
	private $nome;

	// Um método
	public function getNome(){
		// Código
	}
}
```

3.4 - Objeto

Um objeto é criado quando instanciamos uma classe. Sintaxe:
- Nome do objeto
- Atribuição com =
- Operador new
- Nome da classe
Veja abaixo:

```
class Pessoa {
        // Propriedades
        private $nome = 'João Brito';
        public $idade = 25;

        // Métodos
        public function getNome(){
           return $this->nome;
        }

        public function setNome($nome){
           $this->nome = $nome;
        }
}

$pessoa = new Pessoa();
```

3.5 - Constantes

Uma constante é mais ou menos como uma variável, pois guarda um valor, mas na verdade é mais como uma função porque uma constante tem seu valor é imutável durante a execução do programa. Depois de declarar uma constante, ela não muda seu valor.

Por convenção as constantes devem usar nomes tudo em maiúsculas: CONSTANTE, MINHA_CONSTANTE;

Declarar uma constante é fácil, como é feito nesta versão de MyClass -

Declaração

Usamos a palavra reservada const seguida do nome, do = e do valor, como abaixo:

```
class MyClass {
  const REQUIRE_MARGIN = 1.7;

  public function __construct($incomingValue) {
  }
   function mostrarConstante() {
      echo  self::REQUIRE_MARGIN . "\n";
   }

}
```

Acessando de fora da classe

print MyClass::REQUIRE_MARGIN;

Acessando de dentro da classe

self::REQUIRE_MARGIN

Observe que para acessar uma constante não precisamos instanciar a classe.

Acessando propriedades da classe

Após criar um objeto de uma classe, podemos acessar suas propriedades e métodos. Para isso usamos o nome do objeto, seguido do operador -> e do nome da propriedade:

```
$idadePedro = $pessoa->idade;
print $idadePedro;
```

Observe que não podemos acessar a propriedade $nome diretamente, pois é private. Somente através do método getNome(), que é um método getter, criado para dar acesso para propriedades privadas.

Acessando métodos da classe

```
$nomePedro = $pessoa->getNome();
print $nomePedro;

$pessoa->setNome('Pedro Alencar');
print $pessoa->getNome();
```

Importante: para acessar propriedades basta entrar seu nome, mas métodos precisam nome mais (), indicando que é um método. Fique atento pois é constante errarmos por esquecer os () no método.

Operador $this

Este é um operador importante na orientação a objetos. Somente é utilizado dentro da classe para se referir a propriedades e métodos. Observe o formato:

```
$this->nome;
$this->getNome;
```

Usa o operador -> seguido do nome da propriedade ou método sem o $.

Pseudo-variável é um nome que será utilizado como se fosse uma variável, porém, que não é verdadeiramente uma variável. Por isso o prefixo **pseudo**, que significa, falso.

Cada classe PHP, possui uma pseudo-variável de nome `$this`. Chamamos a mesma de pseudo-variável até porque, a mesma funciona de maneira diferente das demais variáveis e quem a manipula, é a máquina virtual do PHP.

O que temos que entender, é que `$this` aponta para um objeto na memória, no caso, o objeto em execução, ou melhor, o objeto onde a mesma está contida, ou então, vamos dizer que a mesma aponta para si própria. Cada instância de classe terá a sua própria pseudo-variável `$this` e esta, apontará para o endereço de memória onde a variável está sendo utilizada.

https://excript.com/php/pseudo-variavel-this-php.html

Convenções de Nomes

Nomes de clases – inicia com maiúscula e CamelCase. Ex: OlaMundo

Nomes de métodos e propriedades – inicial minúsculas e camelCase. Ex: olaMundo

Exemplos simples de classes

```
// Tradicional classe inicial - olaMundo
class OlaMundo {
	function OlaMundo(){
		return "Olá Mundo do PHPOO!";
	}
}

$ola = new OlaMundo();
print $ola->OlaMundo();

// Classe Pessoa
class Pessoa {
	private $nome;
	function setNome($nome){
		$this->nome = $nome;
	}

	function getNome(){
		return $this->nome;
	}
}

$joao = new Pessoa();
$joao->setNome("João Brito");
$pedro = new Pessoa();
$pedro->setNome("Pedro Ribeiro");

print '<b><br><br>Classe Pessoa:<br></b>';
print $joao->getNome();
print '<br>';
print $pedro->getNome();

// Controle de acessos
class Acessos{
	public $variavelPublic = "Variável Pública<br>";
	protected $variavelProtected = "Variável Protegida<br>";
	private $variavelPrivate = "Variável Privada<br>";

	public function getPublic(){
		return $this->variavelPublic;
	}

	public function getProtected(){
		return $this->variavelProtected;
	}

	public function getPrivate(){
		return $this->variavelPrivate;
	}

	public function getMetodoPrivate(){
```

```
                return Acessos::getPrivate();
        }

}

$especificacaoAcesso = new Acessos();
echo $especificacaoAcesso->getPublic();
echo $especificacaoAcesso->getMetodoPrivate();
//echo $especificaAcesso->getPrivate(); // Dará um erro fatal

// Interfaces
interface IPessoa{
   public function setNome($nome);
   public function getNome();
}

interface IPessoaFisica{
   public function setCpf($cpf);
   public function getCpf();
}

interface IPessoaJuridica{
   public function setCnpj($cnpj);
   public function getCnpj();
}

class ClassePessoa implements IPessoa, IPessoaFisica, IPessoaJuridica{
   function __construct($nome, $cpf, $cnpj){
      ClassePessoa::setNome($nome);
      ClassePessoa::setCpf($cpf);
      ClassePessoa::setCnpj($cnpj);
   }

   /* Métodos Set */
   public function setNome($nome){
      $this->nome = $nome;
   }
   public function setCpf($cpf){
      $this->cpf = $cpf;
   }
   public function setCnpj($cnpj){
      $this->cnpj = $cnpj;
   }
   /* Métodos Get */
   public function getNome(){
      return $this->nome;
   }
   public function getCpf(){
      return $this->cpf;
   }
   public function getCnpj(){
      return $this->cnpj;
   }
   function __destruct(){
      echo ClassePessoa::getNome()."<br>".ClassePessoa::getCpf()."<br>".ClassePessoa::getCnpj();
   }
}

$classePessoa = new ClassePessoa("Rodrigo", "324.541.588-98", "6545.2101/0001");
```

4 – Visibilidade/Acesso

A visibilidade de uma propriedade ou método pode ser definida prefixando a declaração com as palavras-chave: 'public', 'protected' ou 'private'. Itens declarados como public podem ser acessados por todo mundo. Protected limita o acesso a classes herdadas (e para a classe que define o item). Private limita a visibilidade para apenas a classe que define o item.

Exemplo

```
<?php

class MinhaClasse
{
   public $publica = 'Public';
   protected $protegida = 'Protected';
   private $privada = 'Private';

   function imprimeAlo()
   {
      echo $this->publica.'<br>';
      echo $this->protegida.'<br>';
      echo $this->privada;
   }
}

$obj = new MinhaClasse();
echo $obj->publica; // Funciona
//echo $obj->protegida; // Erro Fatal
//echo $obj->privada; // Erro Fatal
//$obj->imprimeAlo(); // Mostra Public, Protected e Private
```

Descomente cada uma para ver o resultado

```
class Pai
{
   protected $name = 'João Brito';
}

class Filho extends Pai
{
   // A propriedade protected na classe pai pode ser acessada pela classe filha
   // Mas não pode ser acessada diretamente por um objeto da classe filha
   // Para isso precisa ser acessada através de um método
   function mostrarName()
   {
      return $this->name;
   }
// https://www.codegrepper.com/code-examples/php/how+to+access+private+and+protected+members+in+php
}

$filho = new Filho();
echo $filho->mostrarName(); // Prints João Brito
```

5 - Construtor e Destrutor

5.1 - Construtor

Este método é opcional, não precisamos declará-lo explicitamente. Precisa ser public, seu nome deve ser __construct() e todo o seu código é automaticamente executado sempre que a classe for instanciada.

O PHP permite aos desenvolvedores declararem métodos construtores para as classes. Classes que tem um método construtor chamam o método a cada objeto recém criado, sendo apropriado para qualquer inicialização que o objeto necessite antes de ser utilizado.

> **Nota**: Construtores pais não são chamados implicitamente se a classe filha define um construtor. Para executar o construtor da classe pai, uma chamada a **parent::__construct()** dentro do construtor da classe filha é necessária. Se a classe filha não definir um construtor, será herdado da classe pai como um método normal (se não foi declarado como privado).

https://www.php.net/manual/pt_BR/language.oop5.decon.php

Construtores são métodos que são automaticamente chamados sempre que uma classe é instanciada.

```
class pessoa {
        private $nome; // var ainda é mantido para compatibilizar, mas devemos evitar seu uso

        function __construct($pessoa_nome) {
                $this->nome = $pessoa_nome;
        }

        function set_nome($novo_nome) {
                $this->nome = $novo_nome;
        }

        function get_nome() {
                return $this->nome;
        }
}
?>
```

```
<?php
class ClasseBase {
  function __construct() {
     print "No construtor da ClasseBase<br>";
  }
}
```

O método construtor obrigatoriamente precisa ser public

```
class Construtor
{
   private $nome = 'João Brito';

   public function __construct(){
      print 'O nome é ' . $this->nome;
   }
}
```

```
$nome = new Construtor();
```

Veja que logo após instanciar o objeto da classe Construtor, automaticamente o método __construct() é executado

```
class Noticia
{
   public $titulo;
   public $texto;

   public function __construct($valorTit, $valorTxt)
   {
      $this->titulo = $valorTit;
      $this->texto = $valorTxt;
   }
```

Método construtor na subclasse NoticiaPrincipal

```
class NoticiaPrincipal extends Noticia
{
   public $imagem;

   public function __construct($valor_tit, $valor_txt, $valor_img)
   {
      parent::__construct($valor_tit, $valor_txt);
      $this->imagem = $valor_img;
   }
}
```

O método construtor da classe Noticia é herdado e executado automaticamente na subclasse NoticiaPrincipal. Porém, as "características específicas de NoticiaPrincipal não serão inicializadas pelo método construtor da classe pai". Outro detalhe importante: caso a subclasse NoticiaPrincipal tenha declarado um método construtor em sua estrutura, este mesmo método da classe Noticia não será herdado. Nesse caso podemos chamar o método construtor da classe Noticia, através de uma chamada específica: parent::__construct()

5.2 – Destrutor

O método destrutor será chamado assim que todas as referências a um objeto particular forem removidas ou quando o objeto for explicitamente destruído ou qualquer ordem na sequência de encerramento.

Como os construtores, destrutores pais não serão chamados implicitamente pelo engine. Para executar o destrutor pai, deve-se fazer uma chamada explicitamente a parent::__destruct() no corpo do destrutor.

No PHP chamar o destrutor é desnecessário, tendo em vista que ao fechar o script/programa ele encerra todas as referências aos objetos.

```
class MinhaClasseDestruivel {
  function __construct() {
     print "No construtor<br>";
     $this->name = "MinhaClasseDestruivel";
  }
```

```
  function __destruct() {
      print "Destruindo " . $this->name . "<br>";
  }
}

$obj = new MinhaClasseDestruivel();
?>

public function __destruct()
{
   echo “Destruindo objeto...”;
}
```

Geralmente o método __destruct() é desnecessário, pois quando o script é fechado o PHP é encessado automaticamente

```
class Destrutor {
  function __construct() {
      print "No construtor<br>";
      $this->name = "ClasseDestrutor";
  }

  function __destruct() {
      print "Destruindo  " . $this->name . "<br>";
  }
}

$objeto = new Destrutor();
```

6 – Herança

Herança - Para herdar de uma classe usamos a palavra reservada extends

Aclasse base é chamada de classe pai
A classe que herda é chamada de filha

Assim fazendo a classe filha herda tudo da classe pai, exceto o que for private

Vejamos alguns exemplos:

Connection -> Model -> ClientesModel

- Model extends Connection
- Então Model pode usar métodos e propriedades de Connection

- ClientesModel extends Model
- Então ClientesConnection pode usar métodos e propriedades de Model
- Obs.: para que uma classe que extendeu outra possa usar métodos e/ou propriedades da classe pai, precisa que a classe pai os tenha definido como protected ou public

Veja como se ganhou com isso: o código foi dividido em 3 classes: Connection, Model e ClientesModel. Se não tivesse feito isso, usado herança, então precisaríamos ter uma grande classe com o código das 3.

Herança

Obs: Se os atributos forem do tipo public, podemos atribuir valores diretamente para eles, sem a necessidade de utilizar os métodos. Para manipularmos variáveis na classe, precisamos usar a variável $this, funções e o separador ->. A classe deve utilizar a variável $this para referenciar seus próprios métodos e atributos.

Herança I

```
class SuperClasse{

   public $a = 'variável $a';

   public function ini(){
      echo "SuperClasse->ini()";
   }
}

class SubClasse extends SuperClasse{

}

$super = new SuperClasse();
$super->ini();
$super->a = "";

$sub = new SubClasse();
$sub->ini();
$sub->a = "qualquer valor";
```

https://excript.com/php/heranca-php.html

Para usar o retorno da classe pai na filha com
$this->ret;

Precisamos puxar o construtor da classe pai na filha:

```
<?php
class Pai{
   private $user = 'root';
   private $pass = 'root';
   private $host = 'localhost';
   private $db = 'testes';
   protected $pdo;
   public $regsPerPage = 10;
   public $linksPerPage = 15;

   public function __construct(){
      try {
         $this->pdo = new PDO('mysql:host='.$this->host.';dbname='.$this->db, $this->user, $this->pass);
         //print 'Conectou';
         return $this->pdo;
      } catch (PDOException $e) {
         print "Error: <br>" . $e->getMessage();
         die();
      }
   }
}
$filha = new Pai();
```

```
<?php

require_once 'Pai.php';

/* Classe que trabalha com um crud, lidando com uma tabela por vez, que é fornecida a cada instância, desde a
conexão com o banco */

class Crud extends Conn
{

         public function __construct(){
      parent::__construct();
         }

   // $this->pdo abaixo vem da Pai
   public function update($id,$nome,$email='',$data_nasc=''){
      $sth = $this->pdo->prepare($sql);
      ...
   }
```

6.1 - Substituição/Overriding

Métodos **Override** (substituição), em programação orientada a objeto, é um recurso de linguagem que permite que uma subclasse ou classe filha possa fornecer uma implementação específica de um método que já é fornecido por uma de suas superclasses ou classes pai
https://desenvolvimentoaberto.org/2014/02/16/classes-inheritance-override-e-this-c-2/

Definição de Method Overriding:

É quando a função da classe base é redefinida com o mesmo nome, assinatura e especificador de acesso (public ou protected) da classe derivada.

Overriding/substituição de métodos

As definições de função nas classes filhas substituem as definições com o mesmo nome nas classes pais. Em uma classe filha, podemos modificar a definição de uma função herdada da classe pai.

No exemplo a seguir, as funções getPrice e getTitle são substituídas para retornar alguns valores.

A razão de fazer isso é para prover funcionalidades adicionais sobre as definidas na classe base.

Exemplo prático: você tem uma classe com nome Bird da qual derivam duas outras classes: Eagle e Swift. A classe Bird tem definidos os métodos defined para eat, fly, etc, mas cada uma desstas classes será especializada para Eagle e Swift e deve ter seu próprio estilo de voar que você precisa override/substituir as funcionalidades de voar.

Veja Passaro:

```
<?php
class Passaro {
	public function voar() {
		echo "O método voar da classe Passaro foi chamado";
	}
}

class Aguia extends Passaro {
	public function voar() {
		echo "O método voar da classe Aguia foi chamado";
	}
}

class Andorinha extends Passaro {
	public function voar() {
		echo "O método voar da classe Andorinha foi chamado";
	}
}

$ag = new Aguia();
$an = new Andorinha();

$ag->voar();
echo "<br>";
$an->voar();

Output:
O método voar da classe Aguia foi chamado
O método voar da classe Andorinha foi chamado
```

No exemplo acima criamos dois objetos das classes Aguia e Andorinha. Cada uma destas classes overridden the method voar() e proverá sua própria implementação do método voar() que deve ser extendido da classe Passaro. A forma como ela estenderá o método voar() da classe Passaro não é chamado da mesma forma em ambas as classes que provem uma nova funcionalidade para o método voar().

http://www.sunilb.com/category/php/php5-oops-tutorials

Importante: caso encontre no código o caractere
→
Substitua por
->

6.2 – Sobrecarga/Overloading

Sobrecarga em PHP provê recursos para criar dinamicamente propriedades e métodos. Estas entidades dinâmicas são processadas por métodos mágicos fornecendo a uma classe vários tipos de ações.

Os métodos de sobrecarga são invocados ao interagir com propriedades ou métodos que não foram declarados ou não são visíveis no escopo corrente. O resto desta seção usará os termos propriedades inacessíveis e métodos inacessíveis para referir-se a esta combinação de declaração e visibilidade.

Todos os métodos de sobrecarga devem ser definidos como `públicos`.

Nota:

A interpretação do PHP de sobrecarga é diferente da maioria das linguagens orientadas a objeto. Sobrecarga, tradicionalmente, provê a habilidade de ter múltiplos métodos com o mesmo nome, mas com quantidades e tipos de argumentos diferentes.

https://www.php.net/manual/pt_BR/language.oop5.overloading.php

```
<?php
class MethodTest
{
   public function __call($name, $arguments)
   {
       // Note: value of $name is case sensitive.
       echo "Calling object method '$name' "
            . implode(', ', $arguments). "\n<br>";
   }

   /**  As of PHP 5.3.0  */
   public static function __callStatic($name, $arguments)
   {
       // Note: value of $name is case sensitive.
       echo "Calling static method '$name' "
            . implode(', ', $arguments). "\n<br>";
   }
}

$obj = new MethodTest;
$obj->runTest('in object context');

MethodTest::runTest('in static context');  // As of PHP 5.3.0
?>
```

7 – Polimorfismo

```
<?php

class Cat {
	function miau()
	{
		print "miau";
	}
}

class Dog {
	function latir()
	{
		print "latir";
	}
}

function printTheRightSound($obj)
{
	if ($obj instanceof Cat) {
		$obj->miau();
	} else if ($obj instanceof Dog) {
		$obj->latir();
	} else {
		print "Error: Passed wrong kind of object";
	}
print "<br>";
}

printTheRightSound(new Cat())."<br>";
printTheRightSound(new Dog());

/* Este exemplo não é extensível, pois se adicionarmos sons de mais animais precisaremos estar repetindo código: else if. */

?>
```

No exemplo dado abaixo, a interface com o nome de Shape confirma todas as classes que a implementam para definir um método abstrato com o nome de calcArea ().

```
interface Shape {
  public function calcArea();
}
```

De acordo, a classe Circle implementa a interface colocando no método calcArea () a fórmula que calcula a área dos círculos.

```
class Circle implements Shape {
  private $radius;

  public function __construct($radius)
  {
    $this->radius = $radius;
  }

  // calcArea calculates the area of circles
  public function calcArea()
  {
```

```
    return $this->radius * $this->radius * pi();
  }
}
```

A classe retângulo também implementa a interface Forma, mas define o método calcArea () com uma fórmula de cálculo adequada para retângulos:

```
class Rectangle implements Shape {
  private $width;
  private $height;

  public function __construct($width, $height)
  {
    $this -> width = $width;
    $this -> height = $height;
  }

  // calcArea calculates the area of rectangles
  public function calcArea()
  {
    return $this -> width * $this -> height;
  }
}
```

Agora, podemos criar objetos das classes concretas:

```
$circ = new Circle(3);
$rect = new Rectangle(3,4);
```

Podemos ter certeza de que todos os objetos calculam a área com o método que tem o nome de calcArea (), seja um objeto retângulo ou um objeto círculo (ou qualquer outra forma), desde que implementem a interface Shape.

Agora, podemos usar os métodos calcArea () para calcular a área das formas:

```
echo $circ -> calcArea();
echo $rect -> calcArea();
```

Resultado:

```
28.274333882308
12
```

8 - Classes Abstratas

O PHP 5 introduz métodos e classes abstratas. Classes definidas como abstratas não podem ser instanciadas, e qualquer classe que contenha ao menos um método abstrato também deve ser abstrata. Métodos são definidos como abstratos declarando a intenção em sua assinatura - não podem definir a implementação.

Ao herdar uma classe abstrata, todos os métodos marcados como abstratos na na declaração da classe pai devem ser implementados na classe filha; adicionalmente, estes métodos devem ser definidos com a mesma (ou menos restrita) visibilidade. Por exemplo, se um método abstrato for definido como protegido, a implementação da função deve ser definida como protegida ou pública, mas não privada. Além disso, a assinatura do método deve coincidir, isso é, as induções de tipo e o número de argumentos exigidos devem ser os mesmos. Por exemplo, se a classe filha define um argumento opcional, e a assinatura do método abstrato não, há um conflito na assinatura. Também se aplica a construtores a partir do PHP 5.4. Em versões anteriores a 5.4, as assinaturas dos construtores poderiam ser diferentes.

Exemplos

```
<?php
abstract class ClasseAbstrata
{
   // Força a classe que estende ClasseAbstrata a definir esse método
   abstract protected function pegarValor();
   abstract protected function valorComPrefixo( $prefixo );

   // Método comum
   public function imprimir() {
      print $this->pegarValor();
   }
}

class ClasseConcreta1 extends ClasseAbstrata
{
   protected function pegarValor() {
      return "ClasseConcreta1";
   }

   public function valorComPrefixo( $prefixo ) {
      return "{$prefixo}ClasseConcreta1";
   }
}

class ClasseConcreta2 extends ClasseAbstrata
{
   protected function pegarValor() {
      return "ClasseConcreta2";
   }

   public function valorComPrefixo( $prefixo ) {
      return "{$prefixo}ClasseConcreta2";
   }
}

$classe1 = new ClasseConcreta1;
$classe1->imprimir();
echo $classe1->valorComPrefixo('FOO_') ."\n";
```

```
$classe2 = new ClasseConcreta2;
$classe2->imprimir();
echo $classe2->valorComPrefixo('FOO_') ."\n";
?>
```

O exemplo acima irá imprimir:

ConcreteClass1
FOO_ConcreteClass1
ConcreteClass2
FOO_ConcreteClass2

Outro

```
<?php
abstract class ClasseAbstrata
{
   // Esse método abstrato apenas define os argumentos requeridos
   abstract protected function prefixName($name);

}

class ClasseConcreta extends ClasseAbstrata
{

   // O método filho pode definir argumentos opcionais não presentes na assitura abstrata
   public function prefixName($name, $separator = ".") {
      if ($name == "Pacman") {
         $prefix = "Mr";
      } elseif ($name == "Pacwoman") {
         $prefix = "Mrs";
      } else {
         $prefix = "";
      }
      return "{$prefix}{$separator} {$name}";
   }
}

$class = new ClasseConcreta;
echo $class->prefixName("Pacman"), "\n";
echo $class->prefixName("Pacwoman"), "\n";
?>
```

O exemplo acima irá imprimir:

Mr. Pacman
Mrs. Pacwoman

https://www.php.net/manual/pt_BR/language.oop5.abstract.php

9 - Classe Final

A palavra-chave final, previne que classes filhas sobrescrevam uma definição prefixada com *final*. Se a própria classe estiver definida como final, ela não pode ser estendida.
https://www.php.net/manual/pt_BR/language.oop5.final.php

Exemplos

```
<?php
class ClasseBase {
  public function teste() {
     echo "ClasseBase::teste() chamado\n";
  }

        // palavra-chave 'final', previne que classes filhas sobrecarreguem um método ou variável
  final public function maisTeste() {
     echo "ClasseBase::maisTeste() chamado\n";
  }
}

class ClasseFilha extends ClasseBase {
  public function maisTeste() {
     echo "ClasseFilha::maisTeste() called\n";
  }
}
// Resulta em erro Fatal: Não pode sobrescrever método final ClasseBase::maisTeste()
?>
```

```
<?php
class ClasseBase {
  public function teste() {
     echo "ClasseBase::teste() chamado\n";
  }

  final public function maisTeste() {
     echo "ClasseBase::maisTeste() chamado\n";
  }
}

class ClasseFilha extends ClasseBase {
  public function maisTeste() {
     echo "ClasseFilha::maisTeste() called\n";
  }
}
// Resulta em erro Fatal: Não pode sobrescrever método final ClasseBase::maisTeste()
?>
```

Exemplo #2 Exemplo de classe Final

```
<?php
final class ClasseBase {
  public function teste() {
     echo "Método ClasseBase::teste() chamado\n";
  }

  // A classe já foi marcada como final, esse final aqui é rendundante.
  final public function maisTeste() {
```

```
        echo "Método ClasseBase::maisTeste() chamado\n";
    }
}

class ClasseFilha extends ClasseBase {
}
// Resulta em erro Fatal: Class ClasseFilha may not inherit from final class (ClasseBase)
?>
```

Exemplo #3 Exemplo de constantes finais (PHP 8.1.0 em diante)

```
<?php
class Foo
{
    final public const X = "foo";
}

class Bar extends Foo
{
    public const X = "bar";
}

// Fatal error: Bar::X cannot override final constant Foo::X
?>
```

10 - Métodos e Propriedades Estáticos

Palavra-Chave 'static'
Dica

Esta página descreve o uso da palavra-chave static na definição de métodos e propriedades estáticas. A palavra-chave static também pode ser utilizada para definir variáveis estáticas e em late static bindings. Veja essas páginas para informações desses outros usos de static.

Declarar propriedades ou métodos de uma classe como estáticos faz deles acessíveis sem a necessidade de instanciar a classe. Um membro declarados como estático não pode ser acessado com um objeto instanciado da classe (embora métodos estáticos podem).

Por compatibilidade com o PHP 4, se nenhuma declaração de visibilidade for utilizada, a propriedade ou método será tratado como se declarado como public.
Métodos estáticos

Como métodos estáticos podem ser chamados sem uma instancia do objeto criada, a pseudo-variável $this não está disponível dentro de um método declarado como estático.

Cuidado

No PHP 5, chamar métodos não estáticos estaticamente gerará um alerta de nível E_STRICT.
Aviso

No PHP 7, chamar métodos não estáticos estaticamente foi depreciado, e gerará um alerta de nível E_DEPRECATED. O suporte a chamada de métodos não estáticos estaticamente pode ser removido no futuro.

```
<?php
// Uso do $this e do self:: com classes mistas: estáticas e não estáticas

class Estatica
{
   public static $nome = "Manoel";

   public $sobrenome = 'Castro';

   public static function getNome(){
      return self::$nome;
   }

   public function getSobre(){
      return $this->sobrenome;
   }
}

print Estatica::getNome();

$sobre = new Estatica();
print '<hr>';
print $sobre->getSobre();


<?php
# noticia_estatica.php
```

```
class Noticia
{
   public static $nome_jornal = 'The Unicamp Post';
   protected $titulo;
   protected $texto;

   public function setTitulo($valor)
   {
      $this->titulo = $valor;
   }

   public function setTexto($valor)
   {
      $this->texto = $valor;
   }

   public function exibeNoticia()
   {
      $ret = "<center>";
      $ret .= "Nome do Jornal: <b>" . self::$nome_jornal . "</b><p>";
      $ret .= "<b>". $this->titulo ."</b><p>";
      $ret .= $this->texto;
      $ret .= "</center><p>";
      return $ret;
   }
}

$titulo = 'Vestibular da Unicamp';
$texto = 'Um dos maiores vestibulares do país tem número recorde de inscritos';

$not = new Noticia;
$not->setTitulo($titulo);
$not->setTexto($texto);
$not->exibeNoticia();
echo "<p>" . Noticia::$nome_jornal .'</p>';

/*
Quando utilizamos o modificador static em atributos, ao invés de serem alocados n atributos para n objetos, é alocado
apenas 1 atributo para n objetos, onde todos os objetos têm acesso ao mesmo atributo.
*/
```

Dentro da classe filha NoticiaPrincipal, a chamada à métodos ou propriedades estáticas da classe pai ficaria da seguinte forma:

Propriedade $nome_jornal sendo chamada pela sub-classe NoticiaPrincipal:

```
<?php
# noticia_estatica.php
include_once('noticia_estatica.class.php');

class NoticiaPrincipal extends Noticia
{
   private $imagem;

   public function setImagem($valor)
   {
      $this->imagem = $valor;
   }

   public function exibeNoticia()
```

```
    {
        $ret = "<center>";
        $ret .= "Nome do Jornal: <b>" . parent::$nome_jornal . "</b><p>";
        $ret .= "<img src=\"". $this->imagem ."\"><p>";
        $ret .= "<b>". $this->titulo ."</b><p>";
        $ret .= $this->texto;
        $ret .= "<p></center>";

        return $ret;
    }
}

$titulo = 'Vestibular da Unicamp';
$texto = 'Um dos maiores vestibulares do país tem número recorde de inscritos';
$imagem = 'img_unicamp.jpg';

$not = new NoticiaPrincipal;
$not->setTitulo($titulo);
$not->setTexto($texto);
$not->setImagem($imagem);
$not->exibeNoticia();
print $not;
```

11 – Namespace

Como funciona o namespace no PHP

O objetivo principal do namespace é o de permitir que tenhamos duas ou mais classes com o mesmo nome sem conflito.

Exemplo:

Criar duas pastas:

classes
models

Criar uma classe Produto em classes e models, em arquivos produto.php

classes/produto.php

```
<?php

class Produto{
    public function mostrarDetalhes(){
        echo 'Detalhes do produto da pasta classes';
    }
}
```

e

models/produto.php

```
<?php

class Produto{
    public function mostrarDetalhes(){
        echo 'Detalhes do produto da pasta models';
    }
}
```

Criar no raiz um arquivo index.php

No index.php incluir as duas classes:

```
require 'classes/produto.php';
require 'models/produto.php';

$produto = new Produto();
```

Acusará erro.

Então adicionar o namespace:

```
<?php
namespace classes;

class Produto{
    public function mostrarDetalhes(){
        echo 'Detalhes do produto da pasta classes';
    }
}
```

e

```
<?php

namespace models;

class Produto{
    public function mostrarDetalhes(){
        echo 'Detalhes do produto da pasta models';
    }
}
```

No index.php

```
$produto = new \classes\Produto();
```

e

```
$produto = new \models\Produto();
```

Sem problemas

Referência

Curso de PHPOO do Node Studio
https://www.youtube.com/watch?v=o2CXLk74ggE&list=PLwXQLZ3FdTVEau55kNj_zLgpXL4JZUg8I&index=13

12 – Traits

PHP implementa uma forma de reutilizar o código chamado Traits.

Usando Traits em PHP

O PHP só permite herança única. Em outras palavras, uma classe pode estender apenas uma outra classe. Mas e se você precisar inclui algo que não pertence à classe pai? Antes do PHP 5.4, você teria que ser criativo, mas em 5.4 Traços foram introduzidos. As características permitem que você basicamente "copie e cole" uma parte de uma classe em sua classe principal.

Traits em PHP são uma forma de reutilizar código e de suprir a falta de herança múltipla até a versão 5.4 do PHP.

Eles permitem a reutilização horizontal de código através de classes independentes.

Então aqui temos MrEd, que já está estendendo Horse. Mas nem todos os cavalos falam, então temos uma Característica para isso. Vamos observe o que isso está fazendo
Primeiro, definimos nossa característica. Podemos usá-lo com autoload e namespaces (veja também Referenciando uma classe ou função em um namespace). Em seguida, incluímos em nossa classe MrEd com a palavra-chave use.

Você notará que MrEd começa a usar as funções e variáveis Talk sem defini-las. Lembre-se do que nós disse sobre copiar e colar? Essas funções e variáveis são todas definidas dentro da classe agora, como se esta classe tivesse os definiu.

As características estão mais intimamente relacionadas às classes abstratas, pois você pode definir variáveis e funções. Você também não pode instanciar uma característica diretamente (ou seja, nova característica ()). Traços não podem forçar uma classe a definir implicitamente uma função como um Uma classe abstrata ou uma interface pode. Os traços são apenas para definições explícitas (uma vez que você pode implementar tantos

Interfaces como você deseja, consulte Interfaces).

Quando devo usar uma característica?

A primeira coisa que você deve fazer, ao considerar uma Característica, é se perguntar esta importante questão

Posso evitar o uso de uma característica reestruturando meu código?

Na maioria das vezes, a resposta será sim. Traits são casos extremos causados por herança única. O tentação de mau uso ou uso excessivo As características podem ser altas. Mas considere que um traço apresenta outra fonte para o seu código, o que significa que há outra camada de complexidade. No exemplo aqui, estamos lidando apenas com 3 classes. Mas Traços significam que agora você pode lidar com muito mais do que isso. Para cada característica, sua classe se torna muito mais difícil
para lidar com isso, já que agora você deve consultar cada Característica para descobrir o que ela define (e, potencialmente, onde um colisão ocorreu, consulte Resolução de Conflitos). Idealmente, você deve manter o mínimo possível de Características em seu código.

Traços para facilitar a reutilização de código horizontal

Exemplo

```
<?php
trait Ola
{
   function digaOla() {
      echo "Olá";
   }
}

trait Mundo
{
   function digaMundo() {
      echo " Mundo";
   }
}

class MeuMundo
{
   use Ola, Mundo;
}

$mundo = new MeuMundo();
echo $mundo->digaOla() . $mundo->digaMundo(); //Olá Mundo
```

Mais detalhes:
https://www.sitepoint.com/using-traits-in-php-5-4/
https://pt.slideshare.net/flaviogomesdasilvalisboa/traits-no-php-54-muito-alm-da-herana
http://www.w3programmers.com/multiple-inheritance-in-php-using-traits/

Traits são um mecanismo de reutilização de código em linguagens de herança única, como o PHP. Um Trait destina-se a reduzir algumas limitações de herança única, permitindo a um desenvolvedor reutilizar livremente conjuntos de métodos em várias classes independentes que vivem em diferentes hierarquias de classes. A semântica da combinação de Traits e classes é definida de uma forma que reduz a complexidade, e evita os problemas típicos associados a herança múltipla e Mixins.

Um Trait é semelhante a uma classe, mas destina-se apenas a agrupar a funcionalidade de uma forma fina e consistente. Não é possível instanciar um traço por si só. É um acréscimo à herança tradicional e permite a composição horizontal do comportamento; ou seja, a aplicação dos membros da classe sem exigir herança.

Traduzido com a versão gratuita do tradutor - www.DeepL.com/Translator

Exemplo

```
<?php
trait ezcReflectionReturnInfo {
   function getReturnType() { /*1*/ }
   function getReturnDescription() { /*2*/ }
}

class ezcReflectionMethod extends ReflectionMethod {
   use ezcReflectionReturnInfo;
   /* ... */
}
```

```
class ezcReflectionFunction extends ReflectionFunction {
    use ezcReflectionReturnInfo;
    /* ... */
}
?>
```

https://www.php.net/manual/pt_BR/language.oop5.traits.php

Usando Traits em PHP

Traits em PHP são uma forma de reutilizar código e de suprir a falta de herança múltipla até a versão 5.4 do PHP.

Eles permitem a reutilização horizontal de código através de classes independentes.

Exemplo

```
<?php
trait Ola
{
    function digaOla() {
        echo "Olá";
    }
}

trait Mundo
{
    function digaMundo() {
        echo " Mundo";
    }
}

class MeuMundo
{
    use Ola, Mundo;
}

$mundo = new MeuMundo();
echo $mundo->digaOla() . $mundo->digaMundo(); //Olá Mundo
```

Mais detalhes:
https://www.sitepoint.com/using-traits-in-php-5-4/
https://pt.slideshare.net/flaviogomesdasilvalisboa/traits-no-php-54-muito-alm-da-herana
http://www.w3programmers.com/multiple-inheritance-in-php-using-traits/

13 - Métodos encadeados

Encadeamento de métodos é uma técnica que possibilita executar diversos métodos em um objeto dentro de uma mesma instrução (geralmente separada por ponto-e-vírgula).

Tais métodos, em geral, possuem algum efeito colateral e não retornam valores significativos.

Entretanto, se a chamada de vários métodos é comum nesse objeto, você pode reescrevê-los de forma que eles retornem `$this` e as chamadas poderão ser encadeadas:

```
class Classe {
    public function executar_isto() {
        //faz algo importante
        return $this;
    }
    public function executar_aquilo() {
        //faz algo mais importante
        return $this;
    }
}
$objeto = new Classe();
$objeto->executar_isto()->executar_aquilo();
```

Pode parecer confuso a princípio, mas depois que se entende que as chamadas são feitas sempre no mesmo objeto, que é retornado por cada método, você percebe que isso é na verdade mais fácil de ler e "limpo" do que repetir o objeto várias vezes.

Note que agora os vários métodos são executados dentro de uma única instrução, isto é, sem necessidade de quebrar as chamadas em várias chamadas separadas por ponto-e-vírgula. Isso permite realizar operações *inline*, por exemplo, em parâmetros de métodos e torna o código menos *verboso*:

Encadeamento de métodos é uma técnica que possibilita executar diversos métodos em um objeto dentro de uma mesma instrução (geralmente separada por ponto-e-vírgula).

Tais métodos, em geral, possuem algum efeito colateral e não retornam valores significativos.
Como funciona

Normalmente você executa diversos métodos de um objeto da seguinte forma:

```
class Classe {
   public function executar_isto() {
      //faz algo importante
   }
   public function executar_aquilo() {
      //faz algo mais importante
   }
}
$objeto = new Classe();
$objeto->executar_isto();
$objeto->executar_aquilo();
```

Repare que esses métodos não retornam valor.

Entretanto, se a chamada de vários métodos é comum nesse objeto, você pode reescrevê-los de forma que eles retornem $this e as chamadas poderão ser encadeadas:

```
class Classe {
   public function executar_isto() {
      //faz algo importante
      return $this;
   }
   public function executar_aquilo() {
      //faz algo mais importante
      return $this;
   }
}
$objeto = new Classe();
$objeto->executar_isto()->executar_aquilo();
```

Pode parecer confuso a princípio, mas depois que se entende que as chamadas são feitas sempre no mesmo objeto, que é retornado por cada método, você percebe que isso é na verdade mais fácil de ler e "limpo" do que repetir o objeto várias vezes.

Note que agora os vários métodos são executados dentro de uma única instrução, isto é, sem necessidade de quebrar as chamadas em várias chamadas separadas por ponto-e-vírgula. Isso permite realizar operações inline, por exemplo, em parâmetros de métodos e torna o código menos verboso:

```
$outro_objeto->algum_metodo_legal(
      $objeto->executar_isto()->executar_aquilo(),
      //outros parâmetros
   );
```

Interfaces fluentes e builder pattern

O encadeamento de métodos sozinho não é algo muito atrativo. Porém, se usado com outros padrões como fluent interfaces e builder pattern, o resultado começa a ficar bem interessante.

Se quiser uma leitura adicional, tenho um artigo chamado Construindo objetos de forma inteligente: Builder Pattern e Fluent Interfaces sobre o assunto, mas com alguns exemplos em Java.
Interfaces fluentes

Basicamente, interfaces fluentes consistem em métodos encadeados cujos nomes são significativos para uma determinada operação. Por exemplo:

```
$aviao->abastecer()->decolar()->voarPara("Disney")->pousar();
```

Algumas APIs vão mais longe e criam quase uma DSL (Domain Specific Languages), como é, por exemplo, o caso da biblioteca PDO, com a qual é possível fazer:

```
$data = $pdo->query('SELECT * FROM TABELA')->fetchAll();
```

Builder pattern

Outra aplicação do encadeamento de métodos é na construção de objetos. Fiz um pequeno exemplo no Ideone para ilustrar.

Suponha que sua loja virtual tenha uma cesta de compras com itens, representada pela classe a seguir:

```
class Cesta {
   private $itens;
   public function __construct(array $itens) {
      $this->itens = $itens;
   }
   public function show() {
      echo "Minha cesta:\n";
      foreach ($this->itens as $item) {
         echo $item."\n";
      }
   }
}
```

Note que a implementação de Cesta é imutável, isto é, uma vez construído o objeto, ele não pode mais ser alterado. Há várias vantagens nisso, mas não vou entrar em detalhes aqui. A ideia é que você precisa passar todos os itens de uma vez.

Para facilitar a construção da Cesta, vamos implementar uma calsse builder:

```
class CestaBuilder {
   private $itens = array();
   static public function create() {
      return new static;
   }
   public function adicionar($item) {
      $this->itens[] = $item;
      return $this;
   }
   public function build() {
      return new Cesta($this->itens);
   }
}
```

O nosso builder permite a composição dos itens em sua própria instância e em algum ponto o método build é chamado para retornar uma instância de Cesta com os itens coletados.

Exemplo de uso:

```
$minha_cesta = CestaBuilder::create()
   ->adicionar("Pão")
   ->adicionar("Queijo")
   ->adicionar("Mortadela")
   ->build();
```

Considerações

Mais uma vez, tudo isso pode parecer um pouco confuso a princípio, mas uma vez que se entende bem tais conceitos, dificilmente você vai querer usar outra coisa.

O uso de métodos encadeados, interfaces fluentes e builders deixa seu código mais limpo e intuitivo.

No caso de uma API, evita você ter que ficar olhando a todo momento na documentação procurando quais métodos deve chamar, já que muitas vezes o auto-completar das IDEs já mostra as possibilidades de uso da classe. Isso fica mais evidente uma vez que se acostume ao uso dos padrões, pois ao usar novas APIs você meio que já sabe o que esperar delas.

O que não é encadeamento de métodos

Há uma forma "falsa" de encadear métodos, que na verdade é uma má prática. Consiste em chamar vários métodos em sequência, aparentemente da mesma forma, mas na verdade acessando vários objetos diferentes.

Por exemplo:

```
$contato = $empresa->getProjeto("foo")->getFuncionario("bar")->getNome();
```

Embora o código acima seja intuitivo e compacto, ele traz riscos que um bom design não deveria trazer.

Cada método retorna um objeto diferente, logo pode ser que em alguma chamada o objeto não seja encontrado e null seja retornado. Um erro irá ocorrer sem possibilidade de tratamento.

Além disso, em geral deve-se evitar que um código qualquer tenha conhecimentos sobre vários níveis de objetos, pois isso aumenta muito o acoplamento.

A alternativa nesse caso é criar um método em Empresa para retornar a informação desejada. Exemplo:

```
$contato = $empresa->getNomeContatoPorProjeto("foo","bar");
```

https://pt.stackoverflow.com/questions/105259/o-que-%C3%A9-encadeamento-de-m%C3%A9todos

Métodos encadeados

Utilizar métodos encadeados é uma maneira de se trabalhar de forma mais limpa e dinâmica, chamando um método após o outro sem a necessidade de chamar o objeto de instância da classe.

Entendendo um pouco como funciona o método $this é possível trabalhar com métodos encadeados facilmente, considere a seguinte classe:

Exemplo

```
<?php

class Calculator
{
  protected $result = 0;

  public function sum($num)
  {
    $this->result += $num;
    return $this;
  }

  public function sub($num)
  {
    $this->result -= $num;
    return $this;
  }

  public function result()
```

```
  {
    return $this->result;
  }
}

$calculator = new Calculator;
echo '10 -5 +3 = '.$calculator->sum(10)->sub(5)->sum(3)->result(); // 8
```

14 - Boas práticas

A melhor fonte de informações atualmente sobre boas práticas para PHP é o PHP-Fig

https://www.php-fig.org/

É um consórcioo formado por representantes dos grandes frameworks PHP, que elabora documentos com recomendações padrões sobre a codificação em PHP. Eles criam as PSR (PHP Standard Recommendations). Atualmente temos várias delas. Veja

https://www.php-fig.org/psr/

A PSR 1 é a norma básica sobre codificação.

Um site responsável por divulgar boas práticas para PHP e que atualmente é muito popular é o PHP do Jeito Certo

http://br.phptherightway.com/

O Básico

Operadores de Comparação

Operadores de Comparação são frequentemente negligenciados em PHP, o que pode levar a muitos resultados inesperados. Um desses problemas decorre de comparações estritas (comparações entre booleanos e inteiros).

```
<?php
$a = 5;   // 5 como inteiro

var_dump($a == 5);       // comparação de valores; retorna true
var_dump($a == '5');     // comparação de valores (ignorando os tipos); retorna true
var_dump($a === 5);      // comparação de tipos e valores (integer vs. integer); retorna true
var_dump($a === '5');    // comparação de tipos e valores (integer vs. string); retorna false

/**
 * Comparações Estritas
 */
if (strpos('testing', 'test')) {    // 'test' é encontrado na posição 0, que é interpretado como o booleano 'false'
    // código...
}

vs.

if (strpos('testing', 'test') !== false) {    // true, já que uma comparação estrita foi feita (0 !== false)
    // código...
}
```

Operadores de Comparação
Tabela de Comparação

Estrutura de Controle
Estruturas Condicionais

Quando as declarações ‘if/else’ são usadas em uma função ou classe, é um equívoco comum pensar que ‘else’ precisa ser usado em conjunto para declarar resultados em potencial. Entretanto se o resultado serve para definir o valor a ser retornado ‘else’ não é necessário já que ‘return’ irá terminar a função, fazendo com que o uso de ‘else’ se torne discutível.

```
<?php
function test($a)
{
   if ($a) {
      return true;
   } else {
      return false;
   }
}

// vs
function test($a)
{
   if ($a) {
      return true;
   }
   return false;    // else não é necessário
}
```

Estrutura Condicionais

Estruturas de Decisão

Estruturas de decisão são uma excelente forma de evitar escrever intermináveis estruturas condicionais, mas existem alguns pontos sobre os quais deve-se ficar atento:

Estruturas de decisão só comparam valores, e não tipos (equivalente a ‘==’)
Elas passam por caso a caso até que uma correspondencia seja encontrada, então default é usado (caso esteja definido)
Sem que haja um ‘break’, elas continuarão a executar cada caso até que encontrem um break/return
Dentro de uma função o uso de ‘return’ remove a necessidade do uso de ‘break’ já que isso encerra essa função

```
<?php
$answer = test(2);    // tanto o código para o 'case 2' quanto para o 'case 3' será executado

function test($a)
{
   switch ($a) {
      case 1:
         // código...
         break;            // break é usado para terminar a estrutura de decisão
      case 2:
         // código...       // sem o break, a comparação ira continuar em 'case 3'
      case 3:
         // código...
         return $result;    // dentro de uma função, 'return' termina essa função
      default:
         // código...
         return $error;
   }
}
```

Estruturas de Decisão
PHP Switch

Namespace Global

Quando estiver usando namespaces você pode reparar que funções internas ficam escondidas por funções que você mesmo escreveu. Para corrigir isso refira a funções globais através do uso de uma contra-barra antes do nome da função.

```
<?php
namespace phptherightway;

function fopen()
{
    $file = \fopen();    // O nome da nossa função é igual a de uma função interna.
                         // Execute a função global através da inclusão de '\'.
}

function array()
{
    $iterator = new \ArrayIterator();    // ArrayIterator é uma classe interna. Usar seu nome sem uma contra-barra
                                         // tentará localizar essa função dentro do namespace
}
```

Espaço Global
Regras Globais

Strings

Concatenação

Se sua linha passar do tamanho recomendado (120 caracteres), considere concatenar sua linha
Para facilitar a leitura é melhor usar operadores de concatenação do que operadores de concatenação e atribuição
Enquanto dentro do escopo original da variável, indente quando a concatenação usar uma nova linha

```
<?php
$a  = 'Multi-line example';    // operador de concatenação e atribuição (.=)
$a .= "\n";
$a .= 'of what not to do';

vs.

$a = 'Multi-line example'      // operador de concatenação (.)
    . "\n"                     // indentando novas linhas
    . 'of what to do';
```

Operadores de Strings

Tipos de Strings

Tipos de string são uma característica constante na comunidade PHP, mas talvez essa seção possa explicar as diferenças entre os tipos de strings e seus usos e benefícios.
Aspas Simples

As aspas simples são utilizadas para indicar uma ‘string literal’. Strings literais não tentam analisar caracteres especiais ou variáveis.

Se estiver usando aspas simples, você pode digitar um nome de variável em uma string assim: 'some $thing' e você verá a saída exata some $thing. Se você estiver usando aspas duplas, o motor da linguagem tentará avaliar a variável “$thing” e então exibirá erros se nenhuma variável for encontrada.

```
<?php
echo 'This is my string, look at how pretty it is.';    // sem necessidade de interpretar uma string simples

/**
 * Saída:
 *
 * This is my string, look at how pretty it is.
 */
```

Aspas Simples

Aspas Duplas

Aspas duplas são o canivete suíço das strings, mas são mais lentas devido a interpretação das strings. Ele não só irá analisar as variáveis como mencionado acima mas também todos os tipos de caracteres especiais, como \n para nova linha, \t para identação, etc.

```
<?php
echo 'phptherightway é ' . $adjective . '.'      // Um exemplo com aspas simples que usa concatenação múltipla para
   . "\n"                                 // variáveis e escapar strings
   . 'I love learning' . $code . '!';

// vs
```

echo "phptherightway is $adjective.\n I love learning $code!" // Em vez de concatenação múltipla, aspas duplas

// nos permitem utilizar strings interpretáveis

Aspas duplas que contém variáveis; Isto é chamado “interpolação”.

```
<?php
$juice = 'plum';
echo "I like $juice juice";    // Output: I like plum juice
```

Quando usando interpolação, são comuns os casos onde a variável pode estar colada com outro caracter. Isso fará com que o PHP não consiga interpretar essa variável pelo fato dela estar sendo camuflada.

Para corrigir esse problema envolva a variável em um par de chaves.

```
<?php
$juice = 'plum';
echo "I drank some juice made of $juices";    // $juice cannot be parsed

// vs

$juice = 'plum';
echo "I drank some juice made of {$juice}s";    // $juice will be parsed
```

```
/**
 * Variáveis complexas também serão interpretadas com o uso de chaves
 */

$juice = array('apple', 'orange', 'plum');
echo "I drank some juice made of {$juice[1]}s";   // $juice[1] will be parsed
```

Aspas Duplas

Sintaxe Nowdoc

A Sintaxe Nowdoc foi introduzida no PHP 5.3 e internamente se comporta da mesma forma que as aspas simples exceto que é adequada para o uso de strings de múltiplas linhas sem a necessidade de concatenação.

```
<?php
$str = <<<'EOD'          // iniciada por <<<
Example of string
spanning multiple lines
using nowdoc syntax.
$a does not parse.
EOD;                     // fechando 'EOD' precisa estar na sua própria linha, e no ponto mais a esquerda

/**
 * Output:
 *
 * Example of string
 * spanning multiple lines
 * using nowdoc syntax.
 * $a does not parse.
 */
```

Sintaxe Nowdoc

Sintaxe Heredoc

A Sintaxe Heredoc se comporta internamente da mesma forma que as aspas duplas exceto que é adequada para o uso de strings de múltiplas linhas sem a necessidade de concatenação.

```
<?php
$a = 'Variables';

$str = <<<EOD            // iniciada por <<<
Example of string
spanning multiple lines
using heredoc syntax.
$a are parsed.
EOD;                     // fechando 'EOD' precisa estar na sua própria linha, e no ponto mais a esquerda

/**
 * Output:
 *
 * Example of string
 * spanning multiple lines
 * using heredoc syntax.
 * Variables are parsed.
 */
```

Sintaxe Heredoc

O que é mais rápido?

Há um mito por aí que usar aspas simples em strings são interpretadas mais rápida do que usar aspas duplas. Isso não é fundamentalmente falso.

Se você estiver definindo uma string única e não concatenar valores ou qualquer coisa complicada, então aspas simples ou duplas serão idênticas. Não será mais rápido.

Se você está concatenando várias strings de qualquer tipo, ou interpolar valores em uma string entre aspas duplas, então os resultados podem variar. Se você estiver trabalhando com um pequeno número de valores, a concatenação é minuciosamente mais rápida. Com um monte de valores, interpolação é minuciosamente mais rápida.

Independentemente do que você está fazendo com strings, nenhum dos tipos vai ter qualquer impacto perceptível sobre a sua aplicação. Tentar reescrever código para usar um ou o outro é sempre um exercício de futilidade, de modo a evitar este micro-otimização, a menos que você realmente compreenda o significado e o impacto das diferenças.
Operadores Ternários

O uso de operadores ternários é uma ótima forma de condensar seu código, mas eles são geralmente usados em excesso. Apesar de operações ternárias poderem ser agrupadas e aconselhado usar uma por linha para aumentar a legibilidade.

```
<?php
$a = 5;
echo ($a == 5) ? 'yay' : 'nay';

// vs

$b = 10;
echo ($a) ? ($a == 5) ? 'yay' : 'nay' : ($b == 10) ? 'excessive' : ':(';    // excesso de agrupamento sacrifica a legibilidade
```

Para usar ‘return’ em um operador ternário utilize a sintaxe correta.

```
<?php
$a = 5;
echo ($a == 5) ? return true : return false;    // esse exemplo irá disparar um erro

// vs

$a = 5;
return ($a == 5) ? 'yay' : 'nope';    // esse exemplo irá retornar 'yay'
```

Note que você não precisa usar um operador ternário para retornar um valor booleano. Um exemplo disto seria.

```
<?php
$a = 3;
return ($a == 3) ? true : false; // esse exemplo irá retornar true ou false se $a == 3

// vs

$a = 3;
return $a == 3; // esse exemplo irá retornar true ou false se $a == 3
```

Isso também pode ser dito para as operações (===, !==, !=, == etc).
Utilizando parênteses com operadores ternários para formato e função

Quando se utiliza um operador ternário, os parênteses podem melhorar a legibilidade do código e também incluir as uniões dentro de blocos de instruções. Um exemplo de quando não há nenhuma exigência para usar de parênteses é:

```
<?php
$a = 3;
return ($a == 3) ? "yay" : "nope"; // vai retornar yay ou nope se $a == 3

// vs

$a = 3;
return $a == 3 ? "yay" : "nope"; // vai retornar yay ou nope se $a == 3
```

O uso de parênteses também nos dá a capacidade de criar união dentro de um bloco de declaração onde o bloco será verificado como um todo. Tal como este exemplo abaixo que retornará verdadeiro se ambos ($a == 3 e $b == 4) são verdadeiras e $c == 5 também é verdadeiro.

```
<?php
return ($a == 3 && $b == 4) && $c == 5;
```

Outro exemplo é o trecho de código abaixo que vai returnar true se ($a != 3 e $b != 4) ou $c == 5.

```
<?php
return ($a != 3 && $b != 4) || $c == 5;
```

Operadores Ternários

Declaração de Variáveis

As vezes programadores tentam tornar seu código mais limpo declarando variáveis predefinidas com um nome diferente. O que isso faz na realidade é dobrar o consumo de memória do script. No exemplo abaixo, digamos que uma string de exemplo contém 1MB de dado válido, copiando a variável você aumenta o consumo de memória durante a execução para 2MB.

```
<?php
$about = 'Uma string com texto bem longo';    // usa 2MB de memória
echo $about;

// vs

echo 'Uma string com texto bem longo';        // usa 1MB de memória
```

15 - Padrões de projeto

Esta apostila é apenas uma introdução ao PHP Orientado a Objetos e este item não terá muita cobertura, mas apenas algumas informações úteis.

https://medium.com/@ellingtonbrambila/o-que-s%C3%A3o-padr%C3%B5es-de-projeto-f16d1faa3e2
http://marcelmesmo.blogspot.com/2011/08/o-que-sao-padroes-de-projeto-design.html

Design Patterns

O que são padrões de projeto

- Solução para problemas comuns de desenvolvimento
- Arquiteturas e design de sucesso reutilizáveis
- Linguagem comum para desenvolvedores

Por que usar padrões de design?

- Ajudá-lo a reutilizar arquiteturas e design de sucesso
- Ajudá-lo a resolver problemas que você nunca viu antes
- Forneça um vocabulário comum
- Deixe você se comunicar de forma rápida e inequívoca
- Tornar um sistema mais reutilizável

Regras de ouro

- Programe para uma interface, não uma implementação
- Algumas linguagens levam essa regra para o próximo nível com Duck Typing
- Favorecer a composição do objeto em vez da herança
- Abordagem de caixa preta
- Delegar

Como selecionar um padrão de projeto

- Considere como criar padrões para resolver problemas de projeto
- Estude como os padrões se inter-relacionam
- Padrões de estudo com o mesmo objetivo
- Examine uma causa de reprojeto
- Considere o que deve ser variável em seu design

Como usar um padrão de projeto

- Leia o padrão uma vez para obter uma visão geral
- Volte e estude as seções Estrutura, Participantes e Colaboração
- Veja a seção Código de Amostra para ver um exemplo concreto do padrão no código
- Escolha nomes para participantes do padrão que sejam significativos no contexto do aplicativo
- Definir as classes
- Definir nomes específicos do aplicativo para operações no padrão
- Implementar as operações para executar as responsabilidades e colaborações nos padrões

Quando você está construindo sua aplicação web é muito útil utilizar padrões de codificação para formar a estrutura do seu projeto. Usar “design patterns” (padrões de projeto) é útil pois eles facilitam bastante na hora de gerenciar seu código e permite que outros desenvolvedores entendam rapidamente como tudo está se encaixando.
Se você utiliza um framework então a maior parte do código de alto nível e a estrutura do projeto serão baseados no framework, ou seja, uma grande parte das decisões de padrões de design do código já foram decididas para você. Mas ainda cabe a você escolher os melhores padrões a seguir no código na hora de utilizá-los no framework. Se, por outro lado, você não estiver utilizando uma framework para construir sua aplicação, então você terá que descobrir os padrões que melhor se encaixam para o tipo e tamanho da aplicação que você está construindo.

http://br.phptherightway.com/pages/Design-Patterns.html

https://github.com/DesignPatternsPHP/DesignPatternsPHP

15.1 - SOLID

SOLID - acrônimo de dos cinco princípios. O solid deve:

S - SRP - Single Responsability Principle 'A classe deve ter apenas um e apenas uma responsabilidade

O - OpenClose - A classe deve estar fechada para alterações e aebrta para extensões

L - Liskov substitution - o objeto de uma subclasse tem que poder substituir um atributo da classe pai. A classe pai deve manter somente o que as classes filhas precisarão.

I - Interface segregation

D - Dependency inversion

- Deve ser um código resistente a alterações
- Tornar fácil manter, adaptar e se ajustar às alterações de escopo
- Seja testável e de fácil entendimento
- Seja extensível para alterações com o menor esforço possível
- Forneça o máximo de reaproveitamento
- Permaneça o máximo de tempo possível em utilização
- Baixo acoplamento

Na programação de computador orientada a objetos, SOLID é um acrônimo mnemônico para cinco princípios de design destinados a tornar os designs de software mais compreensíveis, flexíveis e de fácil manutenção. Os princípios são um subconjunto de muitos princípios promovidos pelo engenheiro de software e instrutor Robert C. Martin.

Single-responsibility principle (SRP)

Uma classe deve ter apenas uma única responsabilidade, ou seja, apenas alterações em uma parte da especificação do software devem ser capazes de afetar a especificação da classe.

O princípio de responsabilidade única (SRP) é um princípio de programação de computador que afirma que cada classe em um programa de computador deve ter responsabilidade sobre uma única parte da funcionalidade desse programa, que deve ser encapsulada. Todos os serviços desse módulo, classe, função ou propriedade devem estar estreitamente alinhados com essa responsabilidade.

Open–closed principle

"Entidades de software ... devem ser abertas para extensão, mas fechadas para modificação."

Liskov substitution principle

"Os objetos em um programa devem ser substituídos por instâncias de seus subtipos sem alterar a exatidão desse programa." Veja também design by contract.

Interface segregation principle

"Muitas interfaces específicas do cliente são melhores do que uma interface de uso geral."

Dependency inversion principle

Deve-se "depender de abstrações, [não] de objetos concretos."

Referências

- https://www.youtube.com/watch?v=899Qa6sQcRc - SOLID com TypeScript

- https://en.wikipedia.org/wiki/Solid

- https://pt.wikipedia.org/wiki/SOLID

Padrão front controller

Analisa o URL do pedido e carrega um controlador específico para realmente lidar com o pedido.

16 - PHP Moderno

Ferramentas que colaboram com um PHP Moderno

- Composer
- Git
- GitHub, GitLab, BitBucket e Packagist
- PSRs - http://www.php-fig.org/
- PHP do Jeito Certo - http://br.phptherightway.com/
- XDebug
- Try catch
- Inúmeros pacotes disponíveis/componentes: Whoops, Migrations (Phinx)
- VSCode
- PHPUnit
- phpDoc
- PHPCodeSniffer
- Padrões de Projeto
- MVC
- Modernos Frameworks
- Modernos CMSs
- PDO
- ORMs
- Segurança
- Servidores tipo VPS
- Virtualização e Conteinerização: Virtualbox, Vagrant, Docker
- SOLID is a mnemonic to remind us of five key principles in good object-oriented software design.
- Novos recursos do PHP 7 e 8

O perigo do extremismo
Um problema com regras e diretrizes na programação é que elas geralmente só servem a um propósito em um contexto específico. Saindo desse contexto, uma boa regra pode se tornar uma regra horrível. De fato, toda boa regra se torna ruim quando levada ao extremo.

O princípio KISS, que é um acrônimo para “Keep It Simple, Stupid”, é um bom e extremamente sábio princípio que geralmente é visto por pessoas experientes como um conselho muito bom a seguir, mas mesmo este grande princípio torna-se um perigo para um projeto, se levado ao extremo. Existe tal coisa como “muito simples” resultando em falta de funcionalidade necessária.

Use os frameworks atuais com moderação. Antes de usar verifique se é adequado.

Não somos obrigados a sempre usar padrões de projetos em nosso código. É importante usar quando eles no trará vantagens e evitar quando eles nos trouxer grande complexidade.

Assim também vale para a orientação a objetos. Alguns pequenos projetos não justifica o uso da POO.

Siga o PHP-FIG mas com critérios, de forma a collher algo de útil para você e sua empresa. Não para procurar fazer tudo que o grupo criou.

Fique sempre atento aos bons comportamentos para deixar seu código e aplicativo mais seguros.

http://br.phptherightway.com/
https://phpthewrongway.com/pt_br/
https://www.freecodecamp.org/news/this-is-what-modern-php-looks-like-769192a1320/
https://www.airpair.com/php/posts/best-practices-for-modern-php-development
https://github.com/odan/learn-php
https://medium.com/@FernandoDebrand/guia-pr%C3%A1tico-do-modern-php-desenvolvimento-e-ecossistema-c9715184e463
https://phpthewrongway.com/pt_br/

Por que usar um Framework

Um framework é um conjunto de códigos comuns abstraídos de vários projetos com o objetivo de prover funcionalidades genéricas em um projeto que utilize esse framework. Através de programação, ele permite a customização dessas funcionalidades para torná-las mais específicas de acordo com a exigência de cada aplicação.
De forma resumida o framework é uma estrutura, uma fundação para você criar a sua aplicação. Em outras palavras o framework te permite o desenvolvimento rápido de aplicações (RAD), o que faz economizar tempo, ajuda a criar aplicações mais sólidas e seguras além de reduzir a quantidade de código repetido.

Os desenvolvedores utilizam frameworks por vários motivos, e o maior deles é para agilizar o processo de desenvolvimento. A re-utilização de código em vários projetos vai economizar muito tempo e trabalho… Isso é garantido pois o framework já traz uma série de módulos pré-configurados (e funcionando) para fazer as mais variadas e comuns tarefas como envio de e-mails, conexão com o banco de dados, sanitização (limpeza) de dados e proteção contra ataques.

Não recomendado para:
- Programadores iniciantes
- Projetos pequenos

Pontos fortes:
- Qualidade do código do aplicativo
- Segurança
- Produtividade
- Fácil manutenção do código
- Documentação e material online fartos
- Pequeno tempo de desenvolvimento

17 – Outros

17.1 - Class Not Fount

O erro abaixo é muito comum. Tanto que compilei diversas respostas encontradas

"Class 'aqui o nome da sua classe' not found" error

De tanto esta mensagem aparecer eu cheguei a pensar em abandonar o uso de namespace e voltar para os requires. Mas eu resolvi fazer uma boa pesquisa e anotar algumas observações que eu sabia que poderiam causar isso. E resultou neste documento que resolvi compartilhar no GitHub. Bem, conhecendo melhor sobre namespace estas mensagens parecerão em menor quantidade.

Quando se está usando o autoload com psr-4 e composer precisamos ficar atentos a alguns detalhes algumas vezes resulta na mensagem de erro:

Fatal error: Uncaught Error: Class '\App\Controllers\ClientesController()' not found in ... line 18
Esta é particularmente a minha classe: \App\Controllers\ClientesController()

Algumas Possibilidades para solucionar o erro e também algumams dicas:

- Verifique es está importando o vendor/autoload
    require_once 'vendor/autoload.php';
- Verificar com cuidado o namespace completo da classe que está acusando erro
- Verificar o composer.json, o path dos namespaces na seção autoload com psr-4, se for o caso. Aqui está assim:
    "App\\": "App/",
    "Core\\": "Core/"
    Namespace usa duas barras \\ contrárias. Diretórios usam apenas uma barra /
- Verificar permissões de arquivos em sistemas UNIX
- Cuidado com o case, pois minúsculas são diferentes de maiúsculas
- Verifique se o namespace está declarado na classe que não está sendo encontrada
- Lembre de a cada alteração do composer.json executar:
    composer dumpautoload
    E por segurança, em caso de problemas execute
- Somente podemos usar/importar
        use App\Controller\ClientesController;
        De uma classe que tenha o namespace definido. Não usamos use em arquivos que não contenham classe.
- Verifique atentamente o case dos diretórios e namespace, especialmente se seu sistema não é case sensitivo.
- Eu estava recebendo o erro de classe not found aqui, depois descobri, que eram os parênteses ao final
    $default = '\\App\\Controllers\\'.ucfirst(DEFAULT_CONTROLLER).'Controller()';
    Após remover os parênteses o erro desapareceu

Verifique se o arquivo existe. Exemplo:

```
if(file_exists('App/Controllers/ClientesController.php')){
        print 'sim';
}else{
        print 'nao';
}
exit;

var_dump(file_exists("App/Controllers/ClientController.php"));
```

Testar se a classe existe. Exemplo:

```
if(class_exists('\\App\\Controllers\\ClientesController')){
        print 'sim';
}else{
        print 'não';
}
exit;
```

Exemplo bem definido de criação de instância com namespace
$clientes = new \App\Controllers\ClientesController();

Lembre que as barras do namespace são o contrário das de diretórios:
use Core\Controller\ClientesController;
Diretório: Core/Controllers/ClientesController

Sugestão: crie os nomes dos namespaces idênticos aos dos diretórios, como:
        "App\\": "App/",
Isso facilita.

Outra dica:

Ao usar um require ou include precisamos incluir a extensão .php ao final.
require_once 'Core/Controllers/ClientesController.php';

Ao usar namespace o use não se inclui a extensão:
use Core\Controller\ClientesController;

Podemos instanciar uma classe diretamente através do seu namespace:

$cli = new \Core\Controller\ClientesController;

E também podemos criar uma string com seu namespace:

```
$default = 'App\\Controllers\\ClientesController';
$cli = new $default;
```

Referência
https://dev.to/dechamp/php---how-to-fix-class--not-found-error-1gp9

17.2 - Escopo no PHPOO

Escopo no PHPOO

No PHPOO existem escopos diferentes do PHP estruturado
- private, protected e public

Quando criarmos uma propriedade numa classe tipo private
private $somenteaqui;

private - Todos os métodos da classe podem acessá-la, para ler seu valor e para alterá-lo assim:
$this->somenteaqui;

Mas somente nesta classe esta propriedade pode ser vista e em nenhuma outra classe mais.

protected - Quando criamos uma propriedade numa classe do tipo protected
protected $familia

Todos os métodos da própria classe podem acessar esta propriedade
$this->familia;

Mas também todas as classes que estendem esta classe terão acesso a esta propriedade.

public - Tanto nesta classe quanto em todas as outras esta propriedade é acessível
$this->publico

As mesmas propriedades valem também para os métodos, pois podemos usar private, public e protected nos métodos e nas propriedades.

Exemplo:

```
class pessoa {
	private $nome;
	public $altura;
	protected $posicao_social;
	private $cpf;

	function __construct($pessoa_nome) {
		$this->nome = $pessoa_nome;
	}

	function set_nome($novo_nome) {
		$this->nome = $novo_nome;
	}

	function get_nome() {
		return $this->nome;
	}
}

$pedro = new pessoa("Pedro Mesquita");
echo "Nome completo do Pedro: " . $pedro->get_nome();

	/* Como $cpf foi declarado private, a linha abaixo deverá gerar um erro.
	Descomente para testar!
	*/
	//echo "Diga-me algo particular: " . $pedro->cpf;
```

Exemplo de Herança

```
<?php
class Veiculos
{
  protected $numRodas;
  protected $numPortas;
  protected $cor;
  protected $fabricante;
  protected $modelo;
  protected $chassi;
  protected $preco;

  protected function mostrarPreco(){
    return $this->preco;
  }

  protected function funcionar(){
    return 'Funcionando ...';
  }

}

// Qualquer classe que extenda esta herdará todos seus métodos e propriedades

require_once 'Motocicletas.php';

$moto = new Motocicletas();

$moto->cor();

require_once 'Testes.php';
$veiculo = new Veiculos();
$veiculo->numRodas;
// Receberá o erro: Fatal error: Uncaught Error: Cannot access protected property Veiculos::$numRoda


<?php
class Motocicletas extends Veiculos
{
  protected function testes(){
    $this->numRodas = 2;
    $this->numPortas = 0;
    $this->cor = 'preta';
    $this->fabricante = 'Honda';
    $this->modelo = 'ML 125';
    $this->preco = 65000;
    print $this->numRodas;
    print '<hr>';
    print $this->mostrarPreco();
  }

  public function cor(){
    $this->cor = 'preta';
    print $this->cor;
  }
}

// Classe que não herda de Veiculos para teste
<?php

class Testes
```

```
{
  public function testes(){
    $this->numRodas = 2;
print '<hr>';
    print $this->numRodas;

  }
}
```

Outro exemplo de herança

```
<?php
// Retangulo é mais genérico, enquanto que quadrado é uma especialização de retângulos
class Retangulos
{
    // Declare  properties
    protected $comprimento = 0;
    protected $largura = 0;

    // Method to get the perimeter
    public function getPerimetro(){
        return (2 * ($this->comprimento + $this->largura));
    }

    // Method to get the area
    public function getArea(){
        return ($this->comprimento * $this->largura);
    }
}
```

```
<?php

require_once 'Retangulos.php';

class Quadrados extends Retangulos
{
    public function __construct(){
      $this->largura = 50;
      $this->comprimento = 30;
    }

    public function largura(){
      return $this->largura;
    }

    public function comprimento(){
      return $this->comprimento;
    }

    public function perimetro(){
        $ret = $this->getPerimetro();
        return $ret;
    }

    public function area(){
      $ret = $this->getArea();
      return $ret;
    }
}

$ret = new Quadrados();
```

```
print 'Largura - '.$ret->largura();
print '<hr>';
print 'Comprimento - '.$ret->comprimento();
print '<hr>';

print 'Perímetro - '.$ret->perimetro();
print '<hr>';
print 'Área - '.$ret->area();

//print $ret->largura;// Acusará erro por ser protected
```

Escopo de propriedades dentro e fora dos métodos/funções

```
<?php
    $globalNome = 'Ribamar';

    function ola() {
        $localNome = ' FS';
        echo 'Olá,'. $localNome.'!<br>';
    }

    ola();
    echo "O valor da \$globalNome é: '$globalNome'<br>";
    echo "O valor da \$localNome é: '$localNome'<br>";// Acusará erro, pois $localNome é visível somente dentro da
função ola()

 function minhaFuncao() {
   global $globalNome;
   return $ret = '$globalNome vale '.$globalNome;
 }
 print minhaFuncao();
?>
```

Explicando SuperGlobais

O PHP fornece um conjunto especial de arrays globais contendo várias informações úteis. Esses arrays são conhecidos como superglobais, porque eles são acessíveis em qualquer lugar no seu código – inclusive dentro de funções – e você nem precisa declará-los como global usando a palavra-chave global.

Arqui está a lista completa de superglobais disponíveis no PHP, a partir da versão 5.3:

$GLOBALS: Contém uma lista de todas as variáveis globais disponíveis no script
$_GET: Contém uma lista de todos os campos do formulário enviado pelo navegador usando o pedido GET.
$_POST: Contém uma lista de todos os campos do formulário enviado pelo navegador usando o pedido POST.
$_COOKIE: Contém uma lista de todos os cookies enviados pelo navegador.
$_REQUEST: Contém todas as chaves e valores das variáveis $_GET, $_GET e $_COOKIEcombinados.
$_FILES: Contém uma lista dos arquivos enviados pelo navegador.
$_SESSION: Permite armazenar e recuperar variáveis na sessão atual do navegador.
$_SERVER: Contém informações do servidor, como nome do arquivo em execução, bem como o endereço IP do navegador.
$_ENV: Contém uma lista de variáveis de ambiente passadas pelo navegador. Podem ser variáveis fornecidas pelo Shell, assim como as variáveis CGI.

Constantes

PHP estruturado
define('NOME', 'valor');

PHP orientado a objetos
const NOME = 'Valor';

Chamando:

echo NOME;

echo NomeClasse::NOME;

Constante mágica	Definição
__LINE__	Conterá o número da linha do script na qual ela foi declarada.
__FILE__	Conterá o caminho completo para o arquivo PHP no qual ela foi declarada.
__DIR__	Conterá o diretório do arquivo no qual ela foi declarada.
__FUNCTION__	Conterá o nome da função na qual ela foi declarada.
__CLASS__	Conterá o nome da classe na qual ela foi declarada.
__METHOD__	Conterá o nome da classe e do método no qual ela foi declarada.
__NAMESPACE__	Conterá o nome namespace no qual ela foi declarada.
__TRAIT__	Conterá o nome do trait no qual ela foi declarada.
NomeDaClasse::class	Conterá o nome completo da classe no qual ela foi declarada.

//Declaração das constantes
define('NOME', 'Ribamar FS'); //Declarada a constante NOME com o valor Alex Sander, do tipo String
define ('ALTURA', 1.70); //Declarada a constante ALTURA com o valor 1.76, do tipo float
define('ATIVO', true); //Declarada a constante ATIVO com o valor true, do tipo boolean

const ATIVO = true; //Declarada em OO a constante ATIVO com o valor true, do tipo boolean

echo ATIVO; /* É impresso o valor 1, como ATIVO é do tipo boolean o PHP o
converte o true para 1 e caso fosse false o valor seria 0 */

//Declaração da classe Empresa
class Empresa {
const NOME_EMPRESA = 'Linux & Cia'; //Declarada a constante NOME_EMPRESA do tipo String
const ANO = 2020; //Declarada a constante ANO com o valor 2020, do tipo int
}

17.3 - Funções anônimas

Funções anônimas, também conhecidas como closures, permitem a criação de funções que não tem o nome especificado. Elas são mais úteis como o valor de parâmetros callable, mas podem ter vários outros usos.

Funções anônimas são implementadas utilizando a classe Closure

Closures também podem ser utilizadas como valores de variáveis; o PHP automaticamente converte expressões assim em instâncias da classe interna Closure. Definindo um closure a uma variável usa a mesma sintaxe que qualquer outra definição, incluindo o ponto-e-vírgula:

Exemplo #2 Exemplo de como definir uma função anônima para uma variável

```
$greet = function($name)
{
    print "Hello ". $name.'<br>';
};

$greet('World');
$greet('PHP');


$message = 'hello';

// Sem "use"
$example = function () {
    global $message;
    var_dump($message);
};
$example();

// Inherit $message
$example = function () use ($message) {
    var_dump($message);
};
$example();

// Herdando valor da variável quando a função é definina,
// não quando é chamada
$message = 'world';
$example();

// Reseta mensagem
$message = 'hello';

// Herdando por referência
$example = function () use (&$message) {
    var_dump($message);
};
$example();

// O valor modificado no escopo pai
// reflete quando a função é chamada
$message = 'world';
$example();

// Closures também aceitam argumentos normais
$example = function ($arg) use ($message) {
    var_dump($arg . ' ' . $message);
};
$example("hello");

// Declaração de tipo de retorno após a instrução 'use'
$example = function () use ($message): string {
    return "hello $message";
```

```
};
var_dump($example());
```

17.4 – PHPDoc

PHPDOC – Documentando bem seu código
por Leo Genilhu

Hoje vamos falar um pouco sobre documentação de códigos php usando a ferramenta PHPDoc ou PHPDocumentor. O PHPDoc foi baseado no JAVADoc da Sun e tem como objetivo padronizar a documentação de códigos PHP. Ele lê o código e analisa gramaticalmente procurando por tags especiais. A partir delas extrai toda documentação usando diferentes formatos (pdf, xml, html, chm Windows help e outros). Todas as tags especiais são escritas dentro do comentários do php /*comentários */ e necessariamente começam com o @ (arroba).

Descrição de algumas tags especiais:

- @access Específica o tipo de acesso(public, protected e private).
- @author Específica o autor do código/classe/função.
- @copyright Específica os direitos autorais.
- @deprecated Específica elementos que não devem ser usados.
- @exemple Definir arquivo de exemplo, $path/to/example.php
- @ignore Igonarar código
- @internal Documenta função interna do código
- @link link do código http://www.exemplo.com
- @see
- @since
- @tutorial
- @name Específica o apelido(alias).
- @package Específica o nome do pacote pai, isto ajuda na organização das classes.
- @param Específica os paramêtros muito usado em funções.
- @return Específica o tipo de retorno muito usado em funções.
- @subpackage Específica o nome do pacote filho.
- @version Específica a versão da classe/função.
- Inline { @internal

Exemplo parte de código documentado com PHPDoc.

```
<?php
/**
 * Comentário de cabeçalho de arquivos
* Esta classe de upload de fotos
*
* @author leo genilhu <leo@genilhu.com>
* @version 0.1
* @copyright  GPL © 2006, genilhu ltda.
* @access public
* @package Infra_Estrutura
* @subpackage UploadGenilhu
* @example Classe uploadGenilhu.
*/


class uploadGenilhu {
  /**
```

```
    * Comentário de variáveis
    * Variável recebe o diretório para gravar as fotos.
    * @access private
    * @name $diretorio
    */
    var $diretorio = "" ;

  /**
    * Função para  gravar imagem em diretório
    * @access public
    * @param String $imagem_nome
    * @param String $diretorio
    * @return void
    */
   function upload_up($imagem_nome, $diretorio)
    {
        $tmp = move_uploaded_file($this->arquivo["tmp_name"], $diretorio);
        return($tmp);
    }
?>
```

Como instalar o PHPDoc?

Há dois metódos oficiais para a instalação do PHPDocumentor, você pode baixar o PHPDoc direto do site sourceforge.net. Para instalar pasta descompactar o arquivo .zip e criar um diretorio de trabalho phpdoc.

Outra forma é usar a versão distribuída no pacote PEAR. Para instalar basta executar o comando abaixo
$ pear install PhpDocumentor

Como gerar relatórios de phpdoc usando o ZEND?

O Editor PHP ZEND vem com uma interface integrada para gerar seus relatórios de documentação. Para isso basta seguir as instruções através do menu tools --> PHPDocumentor.
Outra forma de gerar a documentação e via shell do linux.
$phpdoc -o HTML:frames:earthli -f arquivo.php -t docs

Mais curiosidades consulte a documentação no site oficial: phpdoc.org.

Um abraço a todos e até aproxima.

Leo Genilhu

Fontes:

- www.phpdoc.org
- http://pear.php.net/package/PHPDoc
- http://sourceforge.net/projects/phpdoc/

Exemplo

```
<?php
/**
* @author Um nome <a.name@example.com>
* @link http://www.phpdoc.org/docs/latest/index.html
* @package helper
*/
class DateTimeHelper
{
   /**
    * @param mixed $anything Tudo que podemos converter para um objeto \
   DateTime
```

```
    *
    * @return \DateTime
    * @throws \InvalidArgumentException
    */
    public function dateTimeFromAnything($anything)
    {
        $type = gettype($anything);
        switch ($type) {
        // Algum código que tenta retornar um objeto \DateTime
    }
    throw new \InvalidArgumentException(
        "Failed Converting param of type '{$type}' to DateTime object"
        );
    }
    /**
    * @param mixed $date Tudo que podemos converter para um objeto \DateTime
    *
    * @return void
    */
    public function printISO8601Date($date)
    {
      echo $this->dateTimeFromAnything($date)->format('c');
    }

    /**
    * @param mixed $date Tudo que podemos converter para um objeto \DateTime
    */
      public function printRFC2822Date($date)
    {
      echo $this->dateTimeFromAnything($date)->format('r');
    }
}
```

17.5 - Precisa dominar

Assuntos que todo desenvolvedor PHP precisa dominar

Ao levantar esse tema pode gerar muitas polêmicas, porque nem todos tem o mesmo ponto de vista e opinião, o que é ótimo porque pode abrir brechas para ótimas discussões de alto nível.

A ideia deste artigo é basicamente passar um pouco da minha visão e experiência focada trabalhando a mais de 8 anos com PHP.

Os temas que levanto são na minha opinião de extrema importância e fazem total sentido no mundo real corporativo. Separo em três partes, a primeira itens que são temas obrigatórios na segunda parte os assuntos que acredito que são recomendados e por último assuntos que são diferenciais na carreira de um(a) programador(a) PHP, caso queria se aprofundar um pouco mais. Vamos lá!

Temas Obrigatórios

Os temas que vamos levantar agora são assuntos obrigatórios, sem os conhecimentos destes assuntos é impossível dizer que é um(a) programador(a) PHP.

Básico

Obviamente é impossível dizer que desenvolve com PHP se não domina nem o básico da linguagem, neste ponto inclui preparar o ambiente, criar variáveis, comentários de código, loops de repetição, condições (if/else/elseif/endif), funções, arrays, operadores matemáticos, formulários e manipular arquivos e diretórios.

Orientação a Objetos

Outro assunto que precisa estar muito presente e ativo na rotina de qualquer desenvolver PHP é dominar 100% a orientação a objetos, este é um assunto extremamente importante e vai servir como pilar para aprofundar em outros temas da linguagem. Precisa saber criar classes, objetos, métodos da classe, transferir dados de um objeto para outro, funções mágicas da orientação a objetos, encapsulamento e polimorfismo.

Namespace

Este é um tema de extrema importância, organizar projetos seguindo a PSR-4 e trabalhar com namespace para não ter problemas com classes com nomes iguais.

DI

Certamente você já em algum momento já trabalhou com DI (Dependency Injection) mesmo sem saber, é um assunto simples, mas ter a ciência deste tema, é uma obrigação de qualquer programador(a) PHP.

PDO

Essa interface de manipulação de banco de dados levou o nível do PHP para um próximo nível, porque essa interface abstrai muito do trabalho, sendo assim fica muito mais flexível trabalhar com diferentes bases de dados utilizando a mesma estrutura.

MVC

Há muito tempo o MVC é um padrão para desenvolvimento Backend, e este é certamente um assunto que não pode falar no leque de conhecimentos. O MVC vai muito além de separar a aplicação em camadas, e sim definir responsabilidades e ter uma estrutura mais robusta e profissional.

COMPOSER

É o gerenciador de dependências do PHP, o composer facilitou bastante a forma de utilizar pacotes de terceiros em suas aplicações. Antes do composer, se tivesse que utilizar algum código externo, teria que: baixar, testar a compatibilidade e etc, já com o composer este processo ficou muito mais simples, porque o próprio gerenciador de dependências faz isso. Outro benefício do composer é que ele tem o autoload para carregar arquivos de classes de um projeto de forma profissional e simples.

Tests

Este é um assunto que muitos programadores se esquivam, mas, trabalhar com tests especialmente em aplicações de grande porte não significa perca de tempo, muito pelo contrário é ganho de produtividade em longo prazo. Porque trabalhar com tests pode prever erros, te ajudar a ter uma

ideia geral da aplicação, fica mais fácil aplicar correções e também incrementar novas funcionalidades, sem quebrar o que já funcionava antes.

PSRs

Este é um tema super essencial, porque as PSRs definem um estilo visual e padrão para as aplicações PHP, seguir essas recomendações ajuda a comunidade manter um código padronizado e certamente muito mais agradável.

Segurança

É muito comum ouvir a seguinte frase “vou desenvolver agora e depois vejo a que estão de segurança”, essa frase é um suicídio, a questão da segurança deve estar em pauta desde o primeiro estágio do desenvolvimento de uma aplicação com PHP. A parte de segurança deve ser pensada em todos os pontos da criação de um projeto, desde o planejamento.

O PHP não é uma linguagem insegura, ou menos segura que outras, mas a falta de habilidades e conhecimentos com os recursos disponíveis cria uma grande margem de aplicações vulneráveis.

Recomendado:

Neste tópico vou listar alguns assuntos que são recomendados, ao dominar estes temas você vai para outro nível, e certamente terá muito mais portas abertas devido as habilidades mencionadas mais adiante.

Web services

Criar APIs Restful é algo que certamente quando estiver desenvolvendo hora ou outra precisará implementar. Por isso aprender a criar Web Services de forma segura trabalhando com os verbos HTTPs corretos podem ser algo muito recomendado.

DDD

É uma técnica para organizar projetos orientados a domínios. Não é considerado ainda um Pattern como o MVC, mas provavelmente no futuro será. Essa forma de organizar projetos torna o processo de organização de um projeto muito mais amigável, especialmente se estiver trabalhando com um projeto grande.

Design Patterns

O PHP diferente do que se pensa não é terra sem lei, sim existem diversos padrões de projetos para criar aplicações mais profissionais. A falta de conhecimento destes temas pode colocar em risco a fama da linguagem, não pelos recursos mais sim pela inabilidade de quem utiliza. Portanto, estudar os padrões da linguagem é um grande diferencial.

Algum Framework

Algumas pessoas se doem quando toca neste assunto de Frameworks, já outros não sabem fazer nada sem um framework. A realidade é que precisa ter um equilíbrio entre usar e não usar.

Particularmente sou muito favorável ao uso de um framework, inclusive utilizo sempre um framework para criar projetos grandes.

Mas, antes de ficar 100% preso a um framework é muito importante ter domínio dos recursos da linguagem, porque assim será mais fácil de ter uma nova opção. Acho muito útil a ideia de criar um mini-framework pessoal para projetos pequenos, porque isso te dará uma ideia de como funciona um framework PHP.

Diferenciais:

Neste tópico de diferenciais qualquer habilidade extra já encaixa como um diferencial a mais, e essa habilidade pode ir desde algo técnico ou algum tema pessoal. Vale desde se comunicar melhor, liderança, organização até conhecimentos de outras linguagens.

Servidores (deploy)

Quanto maior o conhecimento e servidores melhor. A realidade é que não basta simplesmente saber criar uma aplicação PHP incrível, é necessário ter um mínimo de conhecimento para criar um ambiente digno para comportar e atender a aplicação. Conhecimentos em Cloud atualmente é um tema indispensável.

JavaScript

Já tentou criar uma aplicação web totalmente sem o uso de JS? Aposto que não.
Até é possível criar uma aplicação pequena, mas se pensarmos em aplicações de médio e grande porte o uso de JS é praticamente impossível.

Não só porque JS é uma linguagem incrível e simples, mas porque dá muito mais interatividade na pagina e obviamente melhora muito a usabilidade e experiência do usuário com a página web.
Portanto, domine JavaScript, e algum dos seus milhares de frameworks (Recomendo o Vue JS).

SQL

Ter conhecimentos de banco de dados também é outro grande diferencial, a realidade é que precisa ter um mínimo de noção para conseguir desenvolver, mas, quanto maior o nível de habilidade com bancos de dados SQL melhor.

NoSQL

Outro diferencial é saber trabalhar com NoSQL, ou seja, bancos de dados não-relacionais. Com esse tipo de arquitetura é possível escalar aplicações de forma inimaginável, portanto este tema também surge como um grande diferencial.

Concorda, discorda? Quer acrescentar algo? Comente!!!

Crédito
- https://blog.especializati.com.br/assuntos-que-todo-desenvolvedor-php-precisa-dominar/

17.6 - POO x Procedural

Comparando Código Orientado a Objetos e Código Procedural
Não há maneira certo ou errada de programar. Isso dito, essa seção mostrar bons argumentos para adotar a abordagem orientada a objetos no desenvolvimento de software, especialmente em aplicações de grande porte.

Razão 1: Facilidade de Implementação

"Embora, inicialmente, pareça complicada, POO provê uma abordagem mais fácil para lidar com dados."

Embora, inicialmente, pareça complicada, POO provê uma abordagem mais fácil para lidar com dados. Uma vez que objetos podem guardar dados internamente, variáveis não precisam ser passadas para funções para que elas funcionem corretamente.

E como várias instâncias de uma mesma classe podem existir simultaneamente, lidar com conjuntos de dados grandes se torna infinitamente mais fácil. Por exemplo, imagine que há informações de duas pessoas sendo processadas ao mesmo tempo em um arquivo.
Elas precisam de nomes, ocupações e idades.

A Abordagem Procedural
Eis a abordagem procedural para nosso exemplo:

```
<?php

function changeJob($person, $newjob)
{
  $person['job'] = $newjob; // Muda o emprego da pessoa
  return $person;
}

function happyBirthday($person)
{
  ++$person['age']; // Adiciona 1 à idade da pessoa
  return $person;
}

$person1 = array(
  'name' => 'Tom',
  'job' => 'Button-Pusher',
  'age' => 34
);

$person2 = array(
  'name' => 'John',
  'job' => 'Lever-Puller',
  'age' => 41
);

// Mostra os valores iniciais das pessoas
echo "<pre>Person 1: ", print_r($person1, TRUE), "</pre>";
echo "<pre>Person 2: ", print_r($person2, TRUE), "</pre>";

// Tom foi promovido e fez aniversário
$person1 = changeJob($person1, 'Box-Mover');
$person1 = happyBirthday($person1);

// John fez aniversário também
$person2 = happyBirthday($person2);

// Mostra os novos valores das pessoas
echo "<pre>Person 1: ", print_r($person1, TRUE), "</pre>";
echo "<pre>Person 2: ", print_r($person2, TRUE), "</pre>";

?>
```

Quando executado, o código mostrará o seguinte:

```
Person 1: Array
(
  [name] => Tom
  [job] => Button-Pusher
  [age] => 34
)
Person 2: Array
(
  [name] => John
  [job] => Lever-Puller
  [age] => 41
)
Person 1: Array
(
  [name] => Tom
  [job] => Box-Mover
  [age] => 35
)
Person 2: Array
(
  [name] => John
  [job] => Lever-Puller
  [age] => 42
)
```

Apesar disso não estar tão ruim, tem muita coisa a ser lembrada de uma só vez. O conjunto de atributos da pessoa que foi afetado tem de ser passado e retornado de cada invocação de função, e isso possibilita o aparecimento de erros.

Para melhorar esse exemplo, seria desejável deixar o mínimo para o programador quanto possível. Só a informação extremamente essencial para a operação atual é necessária ser passada para as funções.

É aqui que a POO entra e ajuda você a ajustar as coisas.

A Abordagem POO

Eis a abordagem POO para nosso exemplo:

```
<?php

class Person
{
  private $_name;
  private $_job;
  private $_age;

  public function __construct($name, $job, $age)
  {
     $this->_name = $name;
     $this->_job = $job;
     $this->_age = $age;
  }

  public function changeJob($newjob)
  {
     $this->_job = $newjob;
```

```
  }

  public function happyBirthday()
  {
     ++$this->_age;
  }
}

// Cria duas novas pessoas
$person1 = new Person("Tom", "Button-Pusher", 34);
$person2 = new Person("John", "Lever Puller", 41);

// Mostra seus valores iniciais
echo "<pre>Person 1: ", print_r($person1, TRUE), "</pre>";
echo "<pre>Person 2: ", print_r($person2, TRUE), "</pre>";

// Tom foi promovido e fez aniversário
$person1->changeJob("Box-Mover");
$person1->happyBirthday();

// John também fez aniversário
$person2->happyBirthday();

// Mostra os valores finais
echo "<pre>Person 1: ", print_r($person1, TRUE), "</pre>";
echo "<pre>Person 2: ", print_r($person2, TRUE), "</pre>";

?>
```

O seguinte seria mostrado no navegador:

```
Person 1: Person Object
(
  [_name:private] => Tom
  [_job:private] => Button-Pusher
  [_age:private] => 34
)

Person 2: Person Object
(
  [_name:private] => John
  [_job:private] => Lever Puller
  [_age:private] => 41
)

Person 1: Person Object
(
  [_name:private] => Tom
  [_job:private] => Box-Mover
  [_age:private] => 35
)

Person 2: Person Object
(
  [_name:private] => John
  [_job:private] => Lever Puller
  [_age:private] => 42
)
```

É necessário um pouco mais de preparação na abordagem orientada a objetos, mas, uma vez definida a classe, criar e modificar pessoas é fácil; a informação de uma pessoa não precisa ser passada ou retornada para os métodos e só a informação essencial é passada para cada método. "POO reduzirá significantemente a carga de trabalho se implementada corretamente."

A primeira vista, a diferença pode não parecer tanta, mas, de acordo com que sua aplicação cresce, POO reduzirá significanetemente a carga de trabalho se implementada corretamente.

Dica — Nem tudo precisa ser implementado em orientação a objetos. Uma função rápida, que lida com algo pequeno em um único lugar dentro da aplicação, não precisa, necessariamente, estar envolta em uma classe. Use seu melhor julgamento quando precisar decidir entre as abordagens orientada a objetos ou procedural.

Razão 2: Melhor Organização
Outro benefício da POO é a facilidade de empacotamento e catalogação. Cada classe, geralmente, é mantida sozinha em seu próprio arquivo e, se uma convenção de nomenclatura for usada, acessar as classes será extremamente fácil.

Assuma que você tem uma aplicação com 150 classes que são invocadas dinamicamente através de um arquivo controlador na pasta raiz da sua aplicação. Todas as 150 classes foram nomeadas como class._nome_da_classe_.inc.php e ficar dentro de um diretório chamado inc da sua aplicação.

O controlador implementa a função __autoload() do PHP para carregar dinamicamente as classes que ele precisa executar, ao invés de incluir todas as 150 classes dentro do controlador ou, até mesmo, usar uma maneira "mais esperta" de incluir esses arquivos no seu código:

```
<?php
 function __autoload($class_name)
 {
    include_once 'inc/class.' . $class_name . '.inc.php';
 }
?>
```

Ao termos cada classe separada em seu próprio arquivo, tornamo-as mais portáteis e fáceis de reusar em novas aplicações, sem precisar copiar e colar para todos os lados.

Razão 3: Manutenção Mais Fácil
Devido a natureza compacta do PHP quando programado corretamente, mudanças no código, geralmente, são mais fáceis de perceber e alterar que em uma implementação procedural gigantesca e spaghetti.

Se um conjunto particular de informação ganha um novo atributo, códigos procedurais podem requerer (no pior dos casos) que o novo atributo seja adicionado a cada funcção que usa tal conjunto.

Uma aplicação POO poderia, potencialmente, ser atualizada, bastando adicionar uma nova propriedade e os métodos relacionados que lidam com tal propriedade a uma classe.
Vários benefícios cobertos nessa seção são produto da combinação de POO com práticas de programação DRY. Claro, é possível criar código procedural que seja fácil de manter e que não cause dores-de-cabeça, porém, é igualmente fácil criar código horroroso usando orientação a objetos. [Pro PHP and jQuery] tentará mostrar a combinação de boas práticas de programação com POO para gerar códigos claros e fáceis de ler e manter.

Resumo
A essa altura, espero que você se sinta confortável com o estilo de programação orientado a objetos. Aprender POO é uma ótima maneira de elevar o seu nível de programação. Quando implementada corretamente, POO gera código fácil de ler, manter e portátil, que salvará você (e os desenvolvedores que trabalham com você) horas de trabalho extra. Você está encucado com algo que não foi apresentado nesse artigo? Você já usa POO e tem algumas dicas para os iniciantes? Compartilhe nos comentários!

Nota do Autor — Esse tutorial é um extrato do livro Pro PHP and jQuery (Apress, 2010).
Seja o primeiro a saber sobre novas traduções–siga @tutsplus_pt no Twitter!

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

17.7 - Programação Funcional

Programação Funcional em PHP

O PHP suporta funções de primeira classe, o que significa que uma função pode ser atribuída a uma variável. Ambas definidas pelo usuário e as funções embutidas podem ser referenciadas como uma variável e invocadas dinamicamente. As Funções podem ser passadas como argumentos para outras funções (um recurso chamado de funções de ordem-superior) e uma função pode retornar outras funções.

Recursão, um recurso que permite uma função chamar ela mesma, é suportado pela linguagem, mas a maior parte do código PHP é focado em iteração.

Funções Anônimas (com suporte para closures) estão presentes desde a versão 5.3 do PHP (2009).

No PHP 5.4 foi adicionada a capacidade de atribuir closures no escopo de um objetos e também melhorou o suporte para funções chamáveis de tal forma podem ser usadas como sinônimo de funções anônimas em quase todos os casos.

O uso mais comum de funções de ordem superior ocorre na implementação do padrão Estratégia (Strategy). A função embutida array_filter pede a entrada de array (data) e uma função (uma estratégia ou um callback) usada como um filtro para cada item do array.

```
<?php
$input = array(1, 2, 3, 4, 5, 6);

// cria uma nova função anônima e atribui a uma variável
$filter_even = function($item) {
   return ($item % 2) == 0;
};

// constrói o array_filter com os dados e a função
$output = array_filter($input, $filter_even);

// a função não precisa ser atribuída a uma variável. Assim também é válido:
$output = array_filter($input, function($item) {
   return ($item % 2) == 0;
});

print_r($output);
```

Uma closure é uma função anônima que pode acessar variáveis importadas a partir de fora do escopo usando qualquer variável global. Teoricamente, um closure é uma função com alguns argumentos fechados (por exemplo, fixo) pelo ambiente quando é definido. Closures podem contornar restrições de escopo de variáveis de uma maneira simples.

No próximo exemplo, usamos closures para definir uma função que retorna um filtro único para array_filter, fora da família de funções de filtro.

```
<?php
/**
 * Cria uma função de filtro anônima que aceita $items > $min
 *
 * Retorna um único filtro fora da família de filtros "maior que n"
 */
function criteria_greater_than($min)
{
   return function($item) use ($min) {
      return $item > $min;
   };
}

$input = array(1, 2, 3, 4, 5, 6);

// Utiliza array_filter em uma entrada com uma função de filtro selecionada
$output = array_filter($input, criteria_greater_than(3));

print_r($output); // itens > 3
```

Cada função de filtro na família aceita apenas elementos maiores que algum valor mínimo. Único filtro retornado pela criteria_greater_than é um closure com o argumento $min pelo valor no escopo (dado como um argumento quando criteria_greater_than é chamada).

A vinculação antecipada é usada por padrão para a importação da variável $mindentro da função criada. Para verdadeiras closures com uma vinculação posterior deve ser usada uma referência quando importar. Imagine um template ou uma biblioteca de validação, onde o closure é definido para capturar variáveis no escopo e acessá-los mais tarde quando a função anônima for compilada.

Leia sobre Funções Anônimas
Mais detalhes em Closures RFC
Leia sobre invocar dinamicamente funções com call_user_func_array

http://br.phptherightway.com/pages/Functional-Programming.html

17.8 - Principais vantagens da POO

A programação orientada a objetos traz uma ideia muito interessante: a representação de cada elemento em termos de um objeto, ou classe. Esse tipo de representação procura aproximar o sistema que está sendo criado ao que é observado no mundo real, e um objeto contém características e ações, assim como vemos na realidade. Esse tipo de representação traz algumas vantagens muito interessantes para os desenvolvedores e também para o usuário da aplicação. Veremos algumas delas a seguir.

A reutilização de código é um dos principais requisitos no desenvolvimento de software atual. Com a complexidade dos sistemas cada vez maior, o tempo de desenvolvimento iria aumentar exponencialmente caso não fosse possível a reutilização. A orientação a objetos permite que haja uma reutilização do código criado, diminuindo o tempo de desenvolvimento, bem como o número de linhas de código. Isso é possível devido ao fato de que as linguagens de programação orientada a objetos trazem representações muito claras de cada um dos elementos, e esses elementos normalmente não são interdependentes. Essa independência entre as partes do software é o que permite que esse código seja reutilizado em outros sistemas no futuro.

Outra grande vantagem que o desenvolvimento orientado a objetos traz diz respeito a leitura e manutenção de código. Como a representação do sistema se aproxima muito do que vemos na vida real, o entendimento do sistema como um todo e de cada parte individualmente fica muito mais simples. Isso permite que a equipe de desenvolvimento não fique dependente de uma pessoa apenas, como acontecia com frequência em linguagens estruturadas como o C, por exemplo.

A criação de bibliotecas é outro ponto que é muito mais simples com a orientação a objetos. No caso das linguagens estruturadas, como o C, temos que as bibliotecas são coleções de procedimentos (ou funções) que podem ser reutilizadas. No caso da POO, entretanto, as bibliotecas trazem representações de classes, que são muito mais claras para permitirem a reutilização.

Entretanto, nem tudo é perfeição na programação orientada a objetos. A execução de uma aplicação orientada a objetos é mais lenta do que o que vemos na programação estruturada, por exemplo. Isso acontece devido à complexidade do modelo, que traz representações na forma de classes. Essas representações irão fazer com que a execução do programa tenha muitos desvios, diferente da execução sequencial da programação estruturada. Esse é o grande motivo por trás da preferência pela linguagem C em hardware limitado, como sistemas embarcados. Também é o motivo pelo qual a programação para sistemas móveis como o Google Android, embora em Java (linguagem orientada a objetos), seja feita o menos orientada a objetos possível.

No momento atual em que estamos, tecnologicamente essa execução mais lenta não é sentida. Isso significa que, em termos de desenvolvimento de sistemas modernos, a programação orientada a objetos é a mais recomendada devido as vantagens que foram apresentadas. Essas vantagens são derivadas do modelo de programação, que busca uma representação baseada no que vemos no mundo real.

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

Neste artigo, quero mostrar algumas diferenças entre a programação estruturada (PE) e a programação orientada a objetos (POO). Logo quando pensamos em criar um programa, temos que analisar qual programação é a mais adequada para um problema em questão. Tanto a programação PE ou a programação OO possuem seus pontos altos e baixos, porém a orientada a objetos tem ganhado a preferência dos desenvolvedores para os novos projetos. Graças a facilidade de manipular os dados pois trabalhamos com classes, objetos, herança, encapsulamento de regras de negócios, estruturas de dados, etc., onde cada classe tem seu objetivo específico, assim fazer qualquer alteração em seu código se torna muito mais fácil, mais rápido e sem “danos” em classes dependentes da mesma.

A programação estruturada é formada apenas por três estruturas, que são sequência, onde uma tarefa é executada logo após a outra, decisão quando um teste lógico é executado ou não, e iteração que a partir do teste lógico algum trecho do código pode ser repetido finitas vezes. Seus códigos ficam em um mesmo bloco, sendo mais difícil e demorado fazer uma alteração, pois teremos que olhar se nenhum outro código depende daquele, fazendo uma análise mais detalhada. É fácil de entender,

sendo usada em cursos introdutórios a programação. No final deste artigo, vemos um exemplo de programação estruturada, um programa bem simples usado somente para exemplo. O mesmo exemplo está disponível para visualização, abaixo do exemplo da PE, onde criamos mais pacotes e classes como boas práticas para programar em OO.

Na imagem da programação estruturada, fica claro quando falamos em declarar variáveis, métodos e funções em uma mesma página. Na orientada a objeto vemos as classes, pacotes, e o que caracteriza uma POO.

Imagine fazer uma alteração em um formulário na programação estruturada, onde teremos que ver o que será afetado, onde teremos que reestruturar e assim olhar código por código e ver se nada foi afetado com a alteração. Na orientada a objetos as rotinas e funções estão em objetos separados, encapsulados, facilitando as alterações e atualizações. Procuramos a classe onde o método esta definido, e assim alterando somente aqueles métodos.

Na fábrica de software nosso foco é Java, e essa linguagem é toda orientada a objetos, pois não existe uma linguagem de PE para Java, onde aprendemos a lidar com classes, métodos, herança, polimorfismo entre outros que caracterizam uma POO. Na minha graduação em Tecnologia em Redes, estamos aprendendo a linguagem C estruturada, onde vemos as vantagens e desvantagens em cima da POO.

Minha opinião é que a melhor forma de programar é a orientada a objeto, pois fica bem mais fácil de manusear os códigos, através da herança podemos usar variáveis e métodos já declaradas em outras classes, fica fácil de dar manutenção no programa, fácil para outros programadores possam entende o raciocínio lógico e também alterar os códigos e em questões de segurança é difícil do código ser copiado graças ao encapsulamento das classes

Crédito
http://fabrica.ms.senac.br/2013/04/programacao-estruturada-versus-programacao-orientada-a-objetos/

17.9 - Novidades do PHP 7 e 8

Novidades da versão 7

PHP nasceu como uma linguagem dinâmica e fracamente tipada e passou a ter melhorias em relação à tipos com a sua evolução.

No início ele aceitava somente typehints de classes e interfaces em argumentos de métodos. Depois, com o lançamento do PHP 7, passou-se a ter suporte a typehints de tips escalares como string, int, float e bool, tipos para retorno de métodos e opção de tipagem estrita com `declare(strict_types=1)`.

No PHP 7.1 passou-se a suportar tipos nullable, ou seja, se a variável aceita nulo ou não e aos tipos void e iterable. E no PHP 7.2 o suporte ao tipo object.

Contudo, o PHP ainda não tinha suporte para tipos em propriedades. Ela quase foi incluída na versão 7.3, mas a equipe de desenvolvimento do core do PHP decidiu incluir com calma e deixaram para o PHP 7.4.

Anteriormente a isso utilizava-se uma annotation em docblock (@var EntityManagerInterface) para sinalizar que determinada propriedade era de um tipo específico. Isso auxiliava IDEs no seu autocomplete e ferramentas de análise estática.

Novidades do PHP 7

Referências
https://tableless.com.br/10-novidades-do-php-7/
https://www.treinaweb.com.br/blog/php-7-e-novidades-do-php-7-1/
https://www.php.net/manual/en/migration70.new-features.php
https://github.com/tpunt/PHP7-Reference
https://devzone.zend.com/4693/php-7-glance/
https://blog.digitalocean.com/getting-ready-for-php-7/
https://www.treinaweb.com.br/blog/novidades-do-php-7-2/
https://imasters.com.br/back-end/php-7-2-quais-sao-as-novidades-da-nova-versao-do-php
https://imasters.com.br/back-end/php-7-3-e-php-8-o-que-esperar-das-proximas-versoes

Novidades do PHP 7.1
- Nullable Types (possibilidade de um parâmetro receber um tipo específico ou null)
- Habilidade de pegar múltiplas exceções num mesmo bloco catch
- Criação de um pseudo-tipo chamado Iterable
- Habilidade de definir visibilidade para constantes de classes (public, private, protected)
- Diversas melhorias à extensão Curl, dentre elas, suporte a HTTP/2 Server Push.
- Tipo void para parâmetros e retornos de funções/métodos.
- Incrementos na utilização list()
- O suporte a mcrypt() foi removido da linguagem (tornou-se defasado devido à implementações mais atualizadas e seguras que hoje temos à disposição).

Capturando múltiplas exceções
Antes a única opção era:

```
try {
    // todo
} catch (MyException1 $e) {
    // todo
} catch (MyException2 $e) {
    // todo
} catch (Exception $e) {
    // todo
}
```

No PHP 7.1 é possível agrupar mais de uma exceção num mesmo bloco catch:

```
try {
    // todo
} catch (MyException1 | MyException2 $e) {
    // todo
} catch (Exception $e) {
    // todo
}
```

Void return

Na 7.1 é possível retornar void como sendo um tipo de retorno válido.

Exemplo:

```
class Squirtle extends Pokemon
{
   public function run() : void
   {
      // todo
   }
}
```

Funções removidas

As funções de acesso a bancos de dados mysql_* (mysql_connect(), mysql_query() entre outras) foram removidas na versão 7. A recomendação agora é usar o PDO.

Funções ereg_* foram removidas. Agora devemos usar uma função preg_*, como preg_match ou preg_replace.

Erros Fatais e Exceções
No PHP 7, erros fatais passaram a ser Exceções. Isso quer dizer que eles podem ser tratados em bloco try/catch, sem interromper a execução do script.

Construtores do PHP 4 continuará sendo possível mas recomendando o __construct();

Definir tipo de retorno para funções:

```
function nomeFuncao() : tipo
{
   // corpo da função
}
```

Por exemplo:

```
function soma($x, $y) : float
{
   return $x + $y + 1.5;
}
```

Criação de classes anônimas

```
function createObject()
{
   return new class{
      public function test()
      {
         echo "test" . PHP_EOL;
      }
   };
}

$obj = createObject();
$obj->test();
```

Abaixo algumas das funcionalidades que se tornarão obsoletas no 7.2:

- __autoload
- $php_errormsg

- create_function()
- func_overload
- (unset) cast
- parse_str() without second argument
- gmp_random()
- each()
- assert() with string argument
- $errcontext argument of error handler

Principais novidades que entrarão no PHP 7.2. Algumas das principais:
- get_class() desabilita o parâmetro nulo
- Impedir number_format() de retornar zero negativo
- Argon2 Password Hash
- Object typehint
- libsodium

Argon2 Password Hash
Argon2 é o algoritmo de hashing de senha vencedor do concurso "Password Hashing Competition" de 2015, sendo assim muito bem recomendado a sua utilização. Ao contrário do Bcrypt, que apenas possui um único fator de custo, o Argon2 é parametrizado por três fatores distintos:
1. Tempo de execução
2. Memória necessária
3. Grau de paralelismo

A função password_hash() é alterada para aceitar PASSWORD_ARGON2I como o algoritmo e aceitar o custo da memória, o custo do tempo e o grau de paralelismo como opções. O exemplo a seguir ilustram a nova funcionalidade:

```
// Argon2i com fatores de custo padrão
password_hash('password', PASSWORD_ARGON2I);

// Argon2i com fatores de custo personalizados
password_hash('password', PASSWORD_ARGON2I, ['memory_cost' => 1<<17, 'time_cost' => 4, 'threads' => 2]);
```

A primeira linguagem de programação a adotar criptografia moderna
Outra novidade que veio na versão 7.2 é referente à criptografia moderna, libsodium, que é parte da extensão principal do PHP 7.2. Sodium é uma biblioteca que facilita a utilização de criptografia, descriptografia, assinaturas, hash de senha e muito mais. Seu objetivo é fornecer todas as operações básicas necessárias para criar ferramentas criptográficas de alto nível.

O PHP continua sendo a linguagem mais popular do lado do servidor para criar sistemas. Com uma participação de mercado estimada em 80%, a linguagem de programação de vinte e poucos anos está em toda parte.

O que se tornará obsoleto no PHP 7.3?
Abaixo, algumas das funcionalidades que se tornarão obsoletas:
- Extensão WDDX;
- Alias de função mbstring usando um prefixo mb_ (por exemplo, mb_ereg);
- mb_detect_encoding() sem strict mode;
- Funções strip_tags() e fgetss();
- Função image2wbmp.

Principais novidades que entrarão no PHP 7.3

- Flexibilidade de sintaxe Heredoc e Nowdoc;
- Permitir vírgula à direita em chamadas de função e método;
- Opção para fazer json_encode e json_decode lançar exceções em erros;
- Atribuições de referências em list();
- Função is_countable().

Novidades sobre o PHP 7.3
https://imasters.com.br/php/php-7-3-conheca-as-novidades-desta-versao

Novidades do PHP 7.4

Poderemos declarar tipos estáticos para variáveis

```
class User
{
   public int $id;
   public string $nome;
   private bool $isAdmin = false;
}
```

Operador null coalescing

```
/ $data['name'] = 'John';

// verifica se a variável foi definida e não é nula utilizando o operador null coalesce
$name = $data['name'] ?? 'anonymous';
echo $name; // anonymous
```

Uma novidade muito forte e muito comentada foi a melhora no desempenho (teve seu motor remodelado), que alguns testes chegou a um ganho de 9 vezes.
Veja aqui - https://rberaldo.com.br/php-7-9-vezes-mais-rapido-que-php-5-6/

https://kinsta.com/pt/blog/php-8/

Union types

Para que o processamento seja exigente e lance mensagens de tipos errados precisamos usar na primeira linha

```
declare(strict_types=1);

public function foo(Foo|Bar $input): int|float;

class Number {
   private int|float $number;

   public function setNumber(int|float $number): void {
      $this->number = $number;
   }

   public function getNumber(): int|float {
      return $this->number;
   }
}
```

```
function myFunction(int|float $number): int
{
  return round($number);
}

function atualizarEstoque(int | float $quantidade)
{
  if ($quantidade > 0) {
    $this->estoque += $quantidade;
  }
}

private int|float $numero;

public function foo(Foo|null $foo): void;
```

No entanto, com o PHP 8.0, eles permitem que você escreva usando argumentos, propriedades e valores de retorno mistos para representar vários desses valores:

- array
- bool
- callable
- int
- float
- null
- object
- resource
- string

mixed is equivalent to the union type array|bool|callable|int|float |object|resource|string|null . Available as of PHP 8.0.0.

Falando de tipos no PHP 8

Ao in vés de usar
union type array|bool|callable|int|float|object|resource|string|null

Agora podemos usar algo equivalente e que simplifica:
mixed

mixed - permitir qualquer tipo

mixed indica que um parâmetro pode aceitar vários (mas não necessariamente todos) os tipos.

gettype(), por exemplo, aceita todos os tipos do PHP, enquanto que a função str_replace() aceita somente strings e arrays.

Podemos usar o mixed em qualquer lugar que usamos tipos: propriedades, parâmetros, retorno

```
class Example {
   public mixed $exampleProperty;
   public function foo(mixed $foo): mixed {}
}

function foo(mixed|null $foo) {}
function foo(?mixed $foo) {}
```

```
function foo($foo): mixed|null {}
function foo($foo): ?mixed {}

public function handle(mixed $request): mixed
{
    // processing request

    return $next($request);
}

function mangleUsers(string|array $users): array
{
    If (is_string($users)) {
        $users = [$users];
    }
    // ...
}

class CloudAttribute {

    protected string $value;

    public function __construct(string $value) {
        $this ->value = $value;
    }

    public function getValue() : string {
        return $this -> value;
    }

}
```

Match expression

Match é uma nova expressão semelhante ao switch e conta com novos recursos. Por ser uma expressão, isso significa que ela conta com a capacidade de retornar valores ou armazená-los em uma variável.

Match suporta apenas expressões de uma linha e não precisa de uma declaração break. Além disso, o Match faz comparações estritas.

```
$result = match (1.0) {
    1.0 => "Float!",
    "foo", "bar" => "foobar!"
};
```

Falando de tipos no PHP 8

union type - podemos usar um ou até todos os tipos existentes
mixed - aceita qualquer tipo existente

Exemplos

```
function mista($nr ): int
{
    return $nr*2;
}

print mista(3.25); // Se amarrarmos em int, retorna apenas 6
```

```
function mista($nr ): int|float
{
   return $nr*2;
}

print mista(3.25);// O union type, aceita tanto o float quanto o int

function mista($nr ): mixed
{
   return $nr*2;
}

print mista(3.25); // O mixed funciona como o union type mas simplifica

function mista($nr ): mixed
{
   return $nr*2;
}

print mista('3.5'); // O mixed funciona como qualquer tipo, em nosso caso inclusive com string

function mista(int|float $nr): int|float
{
   return $nr*2;
}
```

print mista('3.5'); // Aqui ele aceita também e retorna 7, mesmo que euu tenha dito para retornar int ou float ele aceita string. Parece que tem efeito apenas de documentação e não amarra por tipos de forma exigente

Mudanças no método Construtor

Se tratando de sintaxe, uma grande mudança que será proveitosa tanto para desenvolvedores iniciantes quanto para quem já utiliza o PHP a mais tempo é as mudanças no método construtor.

Na programação Orientada a Objetos, o método construtor é um dos mais importantes para uma classe. Portanto, o PHP 8 se propõe a simplifica-lo.

Vamos utilizar como exemplo a criação de uma classe “Pessoa”. Dentro dessa classe, armazenaremos informações de nome, idade e altura. Atualmente, no php 7.4, você precisa fazer o código como no exemplo abaixo:

```
class pessoa{
   public string $nome;
   public int $idade;
   public float $altura;

   public function __construct(
      string $nome,
      int $idade,
      float $altura
   ){
      $this->nome= $nome;
      $this->idade= $idade;
      $this->altura= $altura;
   }
}
```

Se você observar bem, perceberá que você precisou repetir o nome dos atributos três vezes. Portanto, isso acaba tornando o código “redundante”. Pensando nisso, no PHP 8, com método contrutor você conseguirá reescrever a mesma classe conforme o exemplo abaixo:

```
class Person {

   public function __construct(
      public string $name,
      public int $age,
      public float $height
   ) {
   }

}
```

Observe que, agora, você poderá escrever a mesma classe porém com bem menos linhas de código. Além disso, torna o código mais simples de se ler e entender. Portanto, as mudanças do método construtor do PHP 8 serão muito interessante para os desenvolvedores.

Porém, é importante que você saiba que essa funcionalidade possuirá algumas exceções. Você apenas conseguirá utilizar esse recurso em métodos construtores para classes não abstratas. Portanto, para entender melhor esse assunto, recomendamos que você leia a RFC sobre a modificação do método construtor.

Novas Funções no PHP 8

Para finalizarmos nossa lista, não podemos deixar de comentar sobre três novas funções que serão acrescentadas ao PHP. São elas:

str_contains
str_starts_with() e str_ends_with()
get_debug_type

A função str_contains

A nova função str_contains permite realizar uma busca dentro de uma string.

Sua síntaxe será como no exemplo abaixo:

str_contains (string $haystack , string $needle) : bool

Esta sintaxe significa que será executado uma verificação para indicar se $needle está presente dentro da string $haystack. Caso sim, ela retornará o valor booleano true. Caso não esteja, então, retornará false.

Portanto, agora podemos utilizar a função str_contains para escrever o código como no exemplo abaixo:

```
$string = 'Frase de exemplo';
$verificar= 'exemplo';

if (str_contains($string, $verificar)) {
        echo "A String foi encontrada";
} else {
        echo "A String não foi encontrada";
}
```

Como você pode perceber, isso tornará a busca dentro de uma string mais legível e menos propenso a erros.

Você pode estar lendo a RFC sobre essa função para poder verificar todas suas características.
As funções str_starts_with() e str_ends_with()

As funções str_starts_with() e str_ends_with() funcionam parecidos com a função anterior, a str_contains. Porém, a diferença é que elas verificam se um string começa ou termina com determinada string.

Sua similaridade com str_contains também se dá pela sintaxe. Veja nos códigos de exemplo abaixo a sintaxe da str_starts_with() e str_ends_with():

str_starts_with (string $haystack , string $needle) : bool
str_ends_with (string $haystack , string $needle) : bool

Portanto, quando você utilizar essa funções, será possível economizar no uso da CPU. Isso acontece pois não será necessário percorrer por toda uma string, a função irá verificar apenas o inicio ou o final.

A função get_debug_type

A nova função que chegará junto ao PHP 8 é a get_debug_type. Com ela, você poderá retornar o tipo de dado de uma variável.

Portanto, você pode perceber que ela é bem parecida com a função já existente gettype(). Porém, a get_debug_type() representa uma melhoria para o PHP, pois ela consegue retornar a verificação de tipo.

Você pode ver na própria RFC as principais diferenças de retornos entre as funções get_debug_type() e gettype()

https://www.homehost.com.br/blog/tutoriais/php/php-8/

Ajustes na concatenação de strings e números

Iremos falar de um dos primeiros conceitos que aprendemos quando estudamos PHP e também um dos conceitos que pode nos causar muita dor de cabeça: a concatenação. Sabemos que o PHP é uma linguagem fracamente tipada e tenta converter números em strings quando usamos o operador . (ponto) em nosso código. Contudo, isso pode nos causar alguns problemas. Vamos ver um exemplo.

Vamos criar duas variáveis numéricas, realizar a soma delas e então exibir o resultado no navegador. Podemos fazer isso por meio do seguinte código:

```
$number1 = 10;
$number2 = 15;

echo "Resultado: " . $number1 + $number2;
```

Quando testamos esse código vemos o seguinte resultado:
Resultado de nosso código PHP no navegador

O resultado está incorreto e um Warning foi retornado. Isso ocorre pois o PHP tentou concatenar a string “Resultado” com o número 10, o que não deu certo. O responsável por fazer com que a conta fosse desconsiderada e a concatenação ocorresse é o que chamamos de precedência de operadores. Ele define a prioridade dos operadores no código. Talvez você lembre desse conceito nas aulas de expressões matemáticas na escola.

Atualmente o operador de concatenação (.) tem uma precedência superior em relação ao operador de soma (+). Para resolver o problema acima, precisaríamos isolar a conta na expressão usando parênteses, da seguinte forma:

echo "Resultado: " . ($number1 + $number2);

Entretanto, pode ser que esqueçamos disso e continuemos tendo o mesmo erro. Pensando em resolver essa pedra no sapato, foi proposto que na versão 8 do PHP o operador de concatenação tenha uma precedência inferior aos operadores aritméticos, e felizmente essa proposta foi aceita. Assim, se executarmos o código inicial usando o PHP 8, veremos o seguinte resultado:

Resultado de nosso código no navegador

Perfeito! A operação foi realizada e a string “Resultado” foi exibida.
https://hcode.com.br/blog/o-que-ha-de-novo-no-php-8-parte-1

Argumentos Nomeados

Agora, com o PHP 8 podemos passar argumentos para uma função em ordem diferente do que foi definido na função, usando nomes.

Quando trabalhamos com funções ou métodos, sempre ouvimos falar de argumentos ou parâmetros. E sabemos que a ordem que informamos esses valores pode fazer toda a diferença no bom funcionamento do código.

A linguagem PHP sempre possuiu argumentos referenciados por meio de sua posição. Mas com a versão 8, isso mudou. Agora possuímos argumentos nomeados. Isso significa que podemos deixar claro qual argumento os valores se referem. Vamos ver um exemplo disso.

Vamos criar uma função que irá retornar um array dos dados informados:

```
function create_user($username = "admin", $password = "admin", $is_admin = true) {
        $data = [
                'username' => $username,
                'password' => $password,
                'admin' => $is_admin
        ];
        return $data;
}
```

Note que essa função possui um valor padrão para todos os argumentos. Imagine que desejamos realizar a chamada dessa função, mas informando apenas o argumento $password. Para fazer isso, basta informar para a função o nome do argumento sem o sinal de $, definir um sinal de : (dois-pontos) e então informar seu valor:

print_r(create_user(password: '123456'));

Esse código definirá o seguinte resultado:

Resultado de nossa função no navegador

Perceba que a informação de senha foi identificada corretamente, mas os outros dois parâmetros receberam seus valores padrão. Por meio dos argumentos nomeados também podemos informar os argumentos na ordem que desejarmos; não precisamos mais ficar presos à ordem que foi definida na função. Por exemplo, se desejarmos informar o terceiro argumento e depois o primeiro, não tem problema:

print_r(create_user(is_admin: 0, username: 'support'));

O resultado desse código será o seguinte:
Resultado de nossa função no navegador

Os dados também são reconhecidos. Impressionante, não é mesmo?

Esse mesmo padrão pode ser usado em funções nativas do PHP. Podemos ver um exemplo disso com a função array_fill(), que é usada para criar um array personalizado com os valores que determinamos. Ele espera três parâmetros: 1) start_index, que representa o índice inicial do nosso array; 2) count, que representa a quantidade de itens que o array possuirá; e 3) value, que é o valor dos itens. Se desejarmos usar os argumentos nomeados com esta função, podemos usar o seguinte código:

print_r(array_fill(value: 'Hcode', start_index: 0, count: 10));

O resultado será o seguinte:
Array gerado pela função array_fill()

Nosso array foi criado com sucesso
https://hcode.com.br/blog/o-que-ha-de-novo-no-php-8-parte-3

Novidades do PHP 7

Estas novidades afetam tanto o interpretador quanto várias extensões e bibliotecas

Algumas:

- Scalar type hint:
string: String
integer: Números inteiros
float: Números em ponto flutuante
bool: Valores boleanos: true ou false

Podemos forçar o interpretador a exigir declaração de tipos ou não inserindo no início do arquivo:

declare(strict_types=0); //evitar type-checking
declare(strict_types=1); // strict type-checking

Exemplo de funções com parêmetros declarando tipos

```
function hint (int $A, float $B, string $C, bool $D)
{
   var_dump($A, $B, $C, $D);
}
```

Para declarar tipos de retorno de funções:

```
function divide(int $A, int $B) : int
{
   return $A / $B;
}
```

Outra novidade do PHP 7 foi a adição de classes anônimas

```
$object = new class {
   public function hello($message) {
      return "Hello $message";
   }
};

echo$object->hello('PHP');

$helper->sayHello(new class {
   public function hello($message) {
      return "Hello $message";
   }
});
```

Operador Coalesce

```
$role = $_GET['role'] ??'guest';

$A = null; // or not set
$B = 10;

echo $A ?? 20; // 20
echo $A ?? $B ?? 30; // 10
```

Operador spaceship

```
operator<=> equivalent
$a < $b($a <=> $b) === -1
$a <= $b($a <=> $b) === -1 || ($a <=> $b) === 0
$a == $b($a <=> $b) === 0
$a != $b($a <=> $b) !== 0
$a >= $b($a <=> $b) === 1 || ($a <=> $b) === 0
$a > $b($a <=> $b) === 1
```

Função de divisão de inteiros

int intdiv(int $dividend, int $divisor)

Let's take a look at the following few examples:

```
intdiv(5, 3); // int(1)
intdiv(-5, 3); // int(-1)
intdiv(5, -2); // int(-2)
intdiv(-5, -2); // int(2)
intdiv(PHP_INT_MAX, PHP_INT_MAX); // int(1)
intdiv(PHP_INT_MIN, PHP_INT_MIN); // int(1)

// following two throw error
intdiv(PHP_INT_MIN, -1); // ArithmeticError
```

intdiv(1, 0); // DivisionByZeroError

Constante tipo array

```
// the define() example
define('FRAMEWORK', [
'version' => 1.2,
'licence' =>'enterprise'
]);

echo FRAMEWORK['version']; // 1.2
echo FRAMEWORK['licence']; // enterprise

// the class const example
class App {
const FRAMEWORK = [
'version' => 1.2,
'licence' =>'enterprise'
];
}
echo App::FRAMEWORK['version']; // 1.2
echo App::FRAMEWORK['licence']; // enterprise
```

Algumas que fram abandonadas/deprecateds

The following functions are no longer available for use:

ereg_replace
ereg
eregi_replace
eregi
split
spliti
sql_regcase

As tags tipo ASP e tipo JavaScript não são mais aceitas

The PHP script and ASP tags are no longer available:

```
<!-- PHP script tag example -->
<script language="php">
// Code here
</script>

<!-- PHP ASP tag example -->
<%
// Code here
%>
<%=$varToEcho; %>
```

E muitas outras novidades foram introduzidas. Vide capítulo 1 do livro

Modular Programming with PHP 7 da PacketPub

Mais novidades sobre o PHP 7 e PHP 8:

Veja o material na pasta Anexos

18 - Exemplos de classes

18.1 – Classe Conta

```
<?php
Class Conta
{
   public $saldo = 0;
   public $titular;

   function depositar($valor){
     $this->saldo += $valor;
   }

   function sacar($valor){
     if(($this->saldo > 0) && ($this->saldo >= $valor))
     {
       $this->saldo -= $valor;
     }else{
       echo "Saldo insuficiente";
     }
   }

   function verSaldo(){
     echo "Saldo Atual:".$this->saldo. "<br>";
   }
 }

  $novaConta = new Conta();
  $novaConta->verSaldo();
  $novaConta->depositar(500);
  $novaConta->verSaldo();
  $novaConta->sacar(150);
  $novaConta->verSaldo();
?>
```

18.2 – Classe de contas usando herança

```
<?php

class Conta
{
   protected $saldo;

   /**
    * Inicializa a conta com um saldo
    */
   public function __construct($valor)
   {
      $this->saldo = $valor;
   }

   /**
    * Deposita um valor na conta
    */
   public function depositar($valor)
   {
      $this->saldo += $valor;
   }
```

```
    /**
     * Saldo
     */
    public function getSaldo()
    {
        return $this->saldo;
    }

}

class ContaCorrente extends Conta
{
    /**
     * Saca um valor da conta
     */
    public function sacar($valor)
    {
        // Verifica se existe saldo para realizar o saque
        if($valor < $this->saldo) {
            $this->saldo -= $valor;
        }
    }
}


class ContaPoupanca extends Conta
{
    private $limiteSaque;

    public function __construct($saldo, $limiteSaque)
    {
        parent::__construct($saldo);
        $this->limiteSaque = $limiteSaque;
    }

    /**
     * Saca um valor da conta
     */
    public function sacar($valor)
    {
        // Verifica se existe saldo para realizar o saque
        if($valor < $this->saldo && $valor < $this->limiteSaque) {
            $this->saldo -= $valor;
        }
    }
}

$cc = new ContaCorrente(1000);
$cc->sacar(300);
print $cc->getSaldo();

$p = new ContaPoupanca(1000, 400);
$p->sacar(200);
//print $p->getSaldo();
```

No exemplo acima criamos duas classes que estendem a classe Conta, ou seja, que herdam as características e métodos da classe pai, podemos observar que definimos uma método sacar com uma uma regra de negócios distinta para cada tipo de conta.

Além disso também criamos um novo atributo para a classe ContaPoupanca, e adicionamos ele a inicialização da classe (método __construct), então chamamos o __construct da classe pai e inicializamos os atributos da classe filha.

https://blog.eximiaweb.com.br/4-conceitos-orientacao-a-objetos/

18.3 – Classe Formas Geométricas com Herança

```
<?php

abstract class Formas
{
   abstract public function calcularArea($lado);
}

class Circulo extends Formas
{
   public function calcularArea($raio)
   {
      return M_PI * $raio;
   }
}

class Quadrado extends Formas
{
   public function calcularArea($lado)
   {
      return $lado * $lado;
   }
}

class Retangulo extends Formas
{
   private $altura = 5;
   public function calcularArea($base)
   {
      return $base * $this->altura;
   }
}

class Triangulo extends Formas
{
   private $altura = 5;
   public function calcularArea($base)
   {
      return ($base * $this->altura)/2;
   }
}

$circulo1= new Circulo();
$cir = $circulo1->calcularArea(5);
print $cir;

echo "<br>";

$quadrado1= new Quadrado();
$qua = $quadrado1->calcularArea(10);
print $qua;

echo "<br>";
```

```
$retangulo1= new Retangulo();
$ret = $retangulo1->calcularArea(5);
print $ret;

echo "<br>";

$triangulo1= new Triangulo();
$tri = $triangulo1->calcularArea(20);
print $tri;

?>
```

18.4 – Classe Conta Abstrata

```
<?php
/*
```

Classes abstratas

Uma classe abstrata é uma classe que não pode ser instanciada como um objeto diretamente. Ela tem que ser estendida por alguma classe concreta, e quando um objeto desta classe for criado, ele herdará métodos e atributos da classe abstrata.
Podemos ter também métodos abstratos em nossas classes.

Métodos Abstratos

Todo método abstrato precisa, obrigatoriamente, ser implementado na classe filha, ou seja, todas as contas, independentemente do tipo devem possuir as operações básicas de saque, depósito, transferência e consulta. Porém, contas de tipos diferentes podem tratar estas ações de formas diferentes. Por exemplo: um depósito em uma conta poupança gera um certo rendimento ao saldo aplicado - para este caso um método abstrato é uma forma de garantir que este método seja implementado na classe ContaPoupança e em todas as outras classes que estende–lás.

```
*/

abstract class Conta
{
   protected $agencia;
   protected $conta;
   protected $saldo;

   public function __construct($agencia, $conta, $saldo)
   {
      $this->agencia = $agencia;
      $this->conta   = $conta;
      if ($saldo >= 0) {
         $this->saldo = $saldo;
      }
   }

   public function getInfo()
   {
      return "Agência: {$this->agencia}, Conta: {$this->conta}";
   }

   public function depositar($quantia)
   {
      if ($quantia > 0) {
         $this->saldo += $quantia;
      }
   }
```

```
    public function getSaldo()
    {
        return $this->saldo;
    }

    abstract function retirar($quantia);
}


class ContaCorrente extends Conta
{
    public function retirar($quantia){
        return $this->saldo = $this->getSaldo() - $quantia;
    }
}

$c = new ContaCorrente('1150', '234.633', 500);
$c->retirar(200);
print 'Saldo de: '.$c->getSaldo();
```

18.5 – Classe Shape Abstrata

```
<?php

abstract class Shape
{
    var $color;
    var $sides;

    function color( )
    {
        return $this->color;
    }

    function sides( )
    {
        return $this->sides;
    }

    abstract function area( );

    function __construct($color, $sides)
    {
        $this->color = $color;
        $this->sides = $sides;
    }
}

class Circle extends Shape
{
    var $radius;

    function area( )
    {
        return 2 * M_PI * $this->radius;
    }

    function __construct($color, $radius)
    {
```

```
        $this->radius = $radius;
        parent::__construct($color, 1);
    }
}


class Rectangle extends Shape
{
    var $width;
    var $height;

    function area( )
    {
        return $this->width * $this->height;
    }

    function __construct($color, $width, $height)
    {
        $this->width = $width;
        $this->height = $height;
        parent::__construct($color, 4);
    }
}


class Triangle extends Shape
{
    // The length of each side
    var $a;
    var $b;
    var $c;

    function area( )
    {
        // Area using Heron's formula
        $s = ($this->a + $this->b + $this->c)/2;
        $area = sqrt(
            $s * ($s - $this->a) * ($s - $this->b) * ($s - $this->c)
        );

        return $area;
    }

    function __construct($color, $a, $b, $c)
    {
        $this->a = $a;
        $this->b = $b;
        $this->c = $c;
        parent::__construct($color, 3);
    }
}
$t = new Triangle("yellow", 2, 3, 4);
$c = new Circle("blue", 5);
$r = new Rectangle("green", 2, 4);

print "Area of our triangle = {$t->area( )} sq units\n<br>";
print "Area of our circle = {$c->area( )} sq units\n<br>";
print "Area of our rectangle = {$r->area( )} sq units\n<br>";

// The following line causes a fatal error because Shape is abstract
//$s = new Shape("green", 5);
/*
```

Area of our triangle = 2.9047375096556 sq units

Area of our circle = 31.415926535898 sq units

Area of our rectangle = 8 sq units
*/
// https://etutorials.org/Programming/PHP+MySQL.+Building+web+database+applications/Chapter+14.+Advanced+Features+of+Object-Oriented+Programming+in+PHP+5/14.3+Abstract+Classes+and+Interfaces/
?>

18.6 – Exemplos de Traits

```
<?php
/*
Traits
Traits, a partir do PHP 5.4, nos proporcionam uma maneira simples e objetiva de reaproveitamento de código, pois são como classes onde usamos a palavra reservada trait, então escrevemos os métodos que queremos. E para usarmos um trait em uma classe usamos a palavra USE.

Uso de Traits
Note que o método getSaldo() foi reescrito dentro do Trait, ou seja, sobrescreverá os métodos da classe base (pai). Podemos ainda usar múltiplos traits em nossas classes

Uso de Múltiplos Traits
Desta vez temos dois traits com nomes diferentes, e note que sobrescrevemos o método getSaldo() novamente no trait consultaExtrato.
*/
 class Conta {

   public $saldo = 0;

   public function getSaldo() {
     return "Saldo Atual: {$this->saldo}";
   }
 }

 trait Acoes {

   public function getSaldo(){
     return "Saldo Disponivel: {$this->saldo}";
   }

   public function depositar($valor){
     $this->saldo += $valor;
   }

   public function sacar($valor){
     if($this->saldo >= $valor){
       $this->saldo -= $valor;
     }
   }
 }

 class ContaCorrente extends Conta{
   use Acoes;
 }

 $o = new ContaCorrente();
```

```
 $o->depositar(500);
 $o->sacar(200);
 $o->getSaldo();

 // Saldo Disponivel: 300
?>

<?php
 class Conta {
  public $saldo = 0;
  public function getSaldo(){
   echo "Saldo Atual: {$this->saldo}";
  }
 }

 trait Acoes {

  public function depositar($valor){
   $this->saldo += $valor;
  }

  public function sacar($valor){
   if($this->saldo >= $valor){
    $this->saldo -= $valor;
   }
  }
 }

 trait consultaExtrato{

  public function getSaldo(){
   echo "Saldo Disponivel para saque:{$this->saldo}<br>";
  }

  public function gerarExtrato($periodo){
   echo "Gerando extrato período $periodo aguarde...";
  }
 }

 class ContaCorrente extends Conta{
  use Acoes, consultaExtrato;
 }

 $o = new ContaCorrente();
 $o->depositar(500);
 $o->sacar(200);
 $o->getSaldo();
 $o->gerarExtrato('20/01/2013');
?>
```

Download desta apostila e de outras

- PHP
- MySQL
- PDO
- MVC
- Laravel
- E outras em elaboração

https://github.com/ribafs/apostilas

Ou na seção Arquivos de alguns grupos do Facebook