Classes e Objetos

# Introdução

O PHP inclui uma modelagem a objetos completa. Entre os recursos disponíveis há: [visibilidade](https://www.php.net/manual/pt_BR/language.oop5.visibility.php), [classes e métodos abstratos](https://www.php.net/manual/pt_BR/language.oop5.abstract.php) e [finais](https://www.php.net/manual/pt_BR/language.oop5.final.php), [métodos mágicos](https://www.php.net/manual/pt_BR/language.oop5.magic.php) adicionais, [interfaces](https://www.php.net/manual/pt_BR/language.oop5.interfaces.php) e [clonagem](https://www.php.net/manual/pt_BR/language.oop5.cloning.php).

O PHP trata objetos da mesma maneira que referências ou manipuladores, significando que cada variável contém uma referência na memória a um objeto ao invés de uma cópia de todo o objeto dentro da memória.

# O básico

## Class

Uma imagem contendo camisa, placar, geladeira

Descrição gerada automaticamenteQuando vamos criar um objeto, primeiro criamos como se fosse um molde de objetos, onde iremos definir as suas propriedades e comportamentos, e com ele, iremos processar dados, modificar, adicionar, e etc.

**Objeto = Instância**

E para isso tuo utilizamos a palavra reserva “class”.

As propriedades são as variáveis e constantes definidas na class, e método é a mesma coisa que função.

Texto

Descrição gerada automaticamente

class + “nome da class” >>>

Assim como cada variável pode ser de um tipo determinado, a classe também pode, e o tipo da classe é o nome que você dá a classe.

Ao criar uma classe, você está criando um novo tipo. Esse tipo tem o nome da classe, e todas as instâncias (objetos) criadas a partir dessa classe pertencem a esse tipo.

Por exemplo, se você define uma classe Animal, o tipo desse objeto será Animal.  
Ela basicamente é um tipo – de estrutura de dados que você define como vai se comportar.

-------

### new

Logotipo

Descrição gerada automaticamente com confiança baixaSempre que chamamos uma class, atribuímos ela a uma variável, então ela se torna um objeto(instância), e podemos apartir, acessar as propriedades e métodos dessa.

Texto

Descrição gerada automaticamente

Utilizamos a palavra reservada “new” para instanciar um novo objeto.

A partir disso, podemos acessar propriedades e métodos da seguinte forma:

Texto

Descrição gerada automaticamente com confiança média

Podemos também acessar propriedades diretas da classe fazendo o mesmo esquema.

Texto

Descrição gerada automaticamente

A partir do momento que atribuímos uma classe a uma variável e ela se torna um objeto, essa variável armazena uma referência na memória em direção á aquela classe, isso acontece para não acabar duplicando o código da classe e o código ficar muito pesado.

### $this

Texto

Descrição gerada automaticamenteSe estivermos codando dentro de uma função e precisarmos chamar uma propriedade da classe fora da função, precisamos então usar a palavra reservada “$this”.

O operador -> é usado para acessar a propriedade do objeto atual, eliminando a necessidade de um $ adicional, que seria redundante. Pois quando utilizamos $this, o PHP já sabe que estamos querendo acessar alguma propriedade da classe.

### Parâmetros de classe

Podemos também passar parâmetros para uma class, através da palavra reservada “\_\_construct”.

“\_\_construct” é basicamente uma função que é iniciada automaticamente assim que o objeto é instânciado.

Essa função construtc é também chamada de “método construtor”, pois como ele serve de base para definir as propriedades que seraõ passadas para a class, ele meio que constrói toda a base da class.

Texto

Descrição gerada automaticamenteDentro dos parâmetros da função \_\_construct, definimos o que queremos obrigatoriamente requerer, funciona basicamente como se fossem parâmetros para uma função, mas nesse caso, é parâmetros para a classe.

Sempre que fomos instâncias um objeto que tenha um “método construtor” passamos os seus parâmetros assim que instanciamos um novo objeto, como acima.

Dentro do método construtor, também podemos processar os parâmetros recebidos, mas vai depender do objetivo do objeto.

Texto

Descrição gerada automaticamente

O oposto do \_\_construtc é o \_\_destruct!

Assim como o construtor \_\_construct é executado quando o objeto é criado, o \_\_destruct é executado quando o objeto é destruído. Ele pode ser útil para liberar recursos, fechar conexões, salvar estados ou executar qualquer operação de "limpeza" necessária antes de o objeto ser completamente removido da memória.

O \_\_destruct funciona como um "limpador" ou "método de encerramento" para o objeto. Ele é chamado automaticamente pelo PHP quando:

* O objeto está prestes a ser removido da memória.
* O script termina.
* O objeto sai de escopo (como ao final de uma função ou bloco de código).

**Como o \_\_destruct Funciona Sem Definição Explícita**

Se você não definir explicitamente um método \_\_destruct em sua classe, o PHP ainda irá **destruir o objeto automaticamente** ao final do seu ciclo de vida, liberando a memória e recursos associados. No entanto:

* **Nenhum código personalizado** será executado, porque não há \_\_destruct definido.
* O PHP apenas "descarta" o objeto da memória, sem realizar nenhuma ação adicional de limpeza (como fechar uma conexão, salvar dados, etc.).

Texto

Descrição gerada automaticamente

Texto

Descrição gerada automaticamente

Ou seja, o método \_\_destruct é usado para implementar uma espécie de **fechamento personalizado** para o objeto. Ele permite que você defina ações específicas que devem ser executadas **automaticamente** quando o objeto está prestes a ser destruído.

### Propriedades e métodos

Propriedades e métodos de classe vivem em "namespaces" separados , de forma que é possível ter uma propriedade e método com mesmos nomes.

Mas não é recomendado fazer isso, simplesmente porque vai dificultar a manutenção e legibilidade.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

**OBS:** Um classe só pode ter 1 construtor e 1 destrutor.

Texto

Descrição gerada automaticamente com confiança média

Uma *classe* representa uma *abstração*, e uma abstração representa uma *simplificação*:

**Classe < abstração < simplificação**

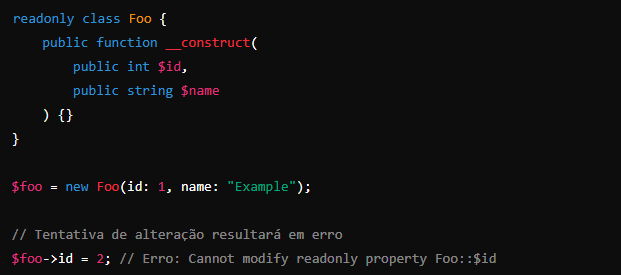
* **Classe como Abstração**: A **classe** é uma forma de abstrair um conceito ou entidade do mundo real. Por exemplo, se você tem uma classe Carro, ela representa a ideia de um carro, com características (como marca, modelo e ano) e comportamentos (como acelerar e frear). A classe abstrai os detalhes específicos de um carro real para definir apenas as informações essenciais que são relevantes no contexto da aplicação.
* **Abstração como Simplificação**: A **abstração** é o ato de simplificar um conceito complexo, incluindo apenas as características e comportamentos relevantes e omitindo detalhes desnecessários. Por exemplo, em uma aplicação de transporte, a classe Carro não precisa ter informações sobre cada componente mecânico detalhado do carro; basta incluir o que é necessário para o objetivo da aplicação, como marca, modelo, e métodos básicos de locomoção. Esse processo de abstrair permite focar no que é importante, eliminando complexidade.
* **Simplificação e Modelagem**: Ao utilizar abstrações, você está simplificando a complexidade do mundo real para criar uma **modelagem** que seja fácil de entender, manipular e implementar. Essa simplificação facilita o desenvolvimento de software, pois permite trabalhar com representações menores e mais manejáveis de conceitos complexos.

Texto

Descrição gerada automaticamente

### readonly Class

A partir do PHP 8.2.0, uma classe pode ser marcada com readonly. Marcar uma classe como readonly irá acrescentar o modificador [readonly](https://www.php.net/manual/pt_BR/language.oop5.properties.php#language.oop5.properties.readonly-properties) em cada propriedade declarada, e prevenir a criação de [propriedades dinâmicas](https://www.php.net/manual/pt_BR/language.oop5.properties.php#language.oop5.properties.dynamic-properties).

Quando você define uma classe como readonly, você está basicamente aplicando um comportamento de "somente leitura" a todas as suas propriedades. Isso garante que, após a construção do objeto, os valores atribuídos a essas propriedades permanecem constantes.

No exemplo acima, após criar o objeto $foo com id e name, qualquer tentativa de alterar o valor de uma dessas propriedades resultará em um erro. Isso oferece segurança adicional, especialmente em cenários onde objetos devem manter seu estado inalterado.

Mas existem outras forma de manter a segurança do código sem usar readonly, que veremos mais a frente.

### ::class

Em PHP, o operador ::class é usado para obter o **nome completo de uma classe**, incluindo o namespace, em forma de string.

Texto

Descrição gerada automaticamenteO ::class é anexado ao nome da classe para gerar uma string com o nome completo da classe. Essa string pode ser usada em vários contextos, como para instanciar objetos dinamicamente, definir tipos em injeção de dependência, ou fazer referência à classe em mensagens de erro ou logs.

### Nullsafe

O **operador nullsafe** (?->) foi introduzido no PHP 8.0 para facilitar o trabalho com objetos que podem ser null. Ele permite acessar métodos e propriedades de objetos de forma segura, evitando erros quando o objeto é null.

Uma imagem contendo Texto

Descrição gerada automaticamenteEm versões anteriores do PHP, para acessar métodos ou propriedades de um objeto que poderia ser null, era necessário usar uma verificação if com is\_null()

Com o operador ?->, você pode acessar métodos e propriedades de um objeto diretamente, e se qualquer parte da cadeia for null, o PHP interrompe a execução da cadeia e retorna null sem lançar uma exceção.



# Propriedades

Variáveis de classes são chamados de *propriedades*. Eles também são conhecidos por outros termos, como *campos* (field).

As variaveis devem ser declaradas inicialmente com 1 dos 3 tipos de visibilidade:

public $var / private $var / protected $var

 **public**: Visível para todos.

 **protected**: Visível apenas para a classe e suas subclasses.

 **private**: Visível apenas dentro da própria classe.

Uma variável que não tiver com a visibilidade definida, fica por padrão como public.

Texto

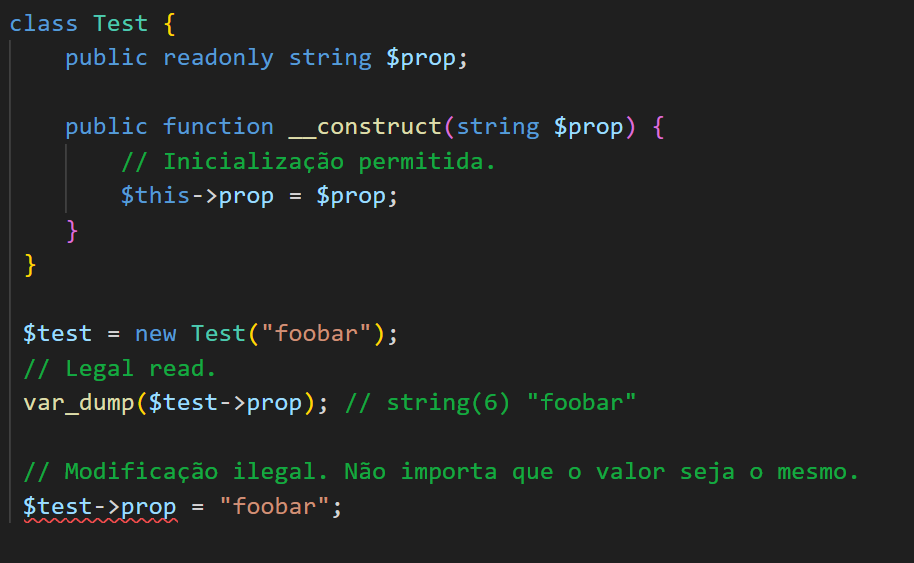
Descrição gerada automaticamenteTamém podemos definir o tipo da variável, como se estivessemos em uma linguagem com tipagem forte.

Em PHP, o ? seguido de um tipo significa que a variável pode ter **dois tipos de valor**: o tipo especificado (neste caso, string) ou null.

### Readonly + propried

Podemos também adicionar o readonly ao declarar propriedades, para que essas fiquem somente em modo leitura.

Antes do PHP 8.4.0, uma propriedade readonly era implicitamente private-set e só poderia ser escrita na mesma classe. A partir do PHP 8.4.0, as propriedades readonly são implicitamente [protected(set)](https://www.php.net/manual/pt_BR/language.oop5.visibility.php#language.oop5.visibility-members-aviz)



O modificador readonly somente pode ser aplicado a [propriedades tipadas](https://www.php.net/manual/pt_BR/language.oop5.properties.php#language.oop5.properties.typed-properties). Uma propriedade somente leitura sem restrições de tipo pode ser declarada utilizando o tipo [Mixed](https://www.php.net/manual/pt_BR/language.types.mixed.php).

Propriedades estáticas não são suportadas com readonly.

Propriedades readonly não podem ter valores padrões pois isso as tornas uma constante, e isso não faz sentido.

A diferença entre readonly e const é que:

Uma propriedade readonly é **imutável após a primeira atribuição**, mas permite que cada instância tenha seu próprio valor, enquanto uma const é uma **constante de classe compartilhada** e **imutável** desde a declaração.

# Constantes de Classe

É possível definir [constantes](https://www.php.net/manual/pt_BR/language.constants.php) em cada classe separadamente. A visibilidade padrão de constantes de classe é public.

O operador :: é o **operador de resolução de escopo**, que é usado para acessar constantes, métodos e propriedades **estáticas** da classe.

Para acessar **constantes** em classes, você deve usar o operador ::, e **não** o operador ->.

Texto

Descrição gerada automaticamente

# Autoloading Classes

O **autoloading de classes** em PHP é um mecanismo que permite carregar automaticamente as classes, interfaces e traits à medida que são necessárias, sem precisar incluir manualmente os arquivos usando include ou require. Com o autoloading, o PHP consegue localizar e carregar as definições de classes dinamicamente, simplificando a organização do código e melhorando a performance.

Texto

Descrição gerada automaticamente

Neste exemplo:

* spl\_autoload\_register registra uma função anônima que tenta incluir o arquivo da classe com base em seu nome.
* Quando $carro = new Carro(); é chamado, o PHP busca o arquivo Carro.php no mesmo diretório.
* Da mesma forma, Usuario.php é carregado quando $usuario = new Usuario(); é instanciado.

Esse exemplo é bastante simples e funciona para projetos pequenos com uma estrutura de arquivos básica, mas não é ideal para projetos maiores.

**OBS:** O spl\_autoload\_register funciona com base em hierarquia de diretórios, ele não sobe.

# Visibilidade

Podemos definir a visibilidade com 3 palavras chaves:

- public

- private

- protected

A função delas é limitar o acesso de propriedades e métodos de alguma classe até certo ponto no código, isso serve para ter mais segurança no código e evitar que dados sensíveis que passam por propriedades e métodos sejam capturados de alguma forma por algum hacker.

**public**:

* Quando um método ou propriedade é declarado como public, ele é acessível de qualquer lugar, ou seja, tanto dentro da própria classe, quanto fora dela e por outras classes que instanciem essa classe. Ele pode ser acessado em instâncias, ou mesmos quando através de seus descendentes.

**private**:

* Um método ou propriedade private só pode ser acessado de dentro da própria classe em que ele foi declarado. Nem mesmo classes filhas (classes que herdam a classe original) têm acesso direto a membros private. Isso é útil para proteger informações ou comportamentos que devem ser encapsulados, ou seja, escondidos de qualquer outra classe.

**protected**:

* A visibilidade protected permite que o método ou propriedade seja acessível apenas dentro da própria classe e em suas subclasses. Diferente do private, ele pode ser acessado em classes que estendem a classe original. Ele é útil para definir atributos e métodos que não devem estar disponíveis para qualquer lugar, mas que ainda podem ser usados ou modificados por classes que derivem da classe atual.

### Visibilidade de Propriedades

Propriedades de classes podem ser declarados como públicas, privadas ou protegidas. Propriedades declaradas sem visibilidade são definidas como públicas.

Como visto anteriormente, as regras de visibilidade padrão vale a mesma paras as propriedades.

**Public:**

* Propriedades public podem ser acessadas e modificadas de qualquer lugar.

**Private:**

* Propriedades private só podem ser acessadas e modificadas apenas dentro da classe.

**Protected:**

* Propriedades protected podem ser acessadas e modificadas apenas por subclasses.

### Visibilidade de métodos

A regra para métodos é a mesma:

**Public:**

* Métodos públicos podem ser acessados de qualquer lugar.

**Private:**

* Métodos privados só podem ser acessados dentro da própria classe.

**Protected:**

* Métodos protegidos só podem ser acessados por subclasses.

### Métodos Getter e Setter

Métodos **getter** e **setter** são usados em programação orientada a objetos para controlar o acesso e a modificação de propriedades de uma classe.

* **Getter**: É um método que **obtém** (ou "pega") o valor de uma propriedade. O termo vem do verbo em inglês "to get", que significa "obter". Normalmente, um método getter apenas retorna o valor da propriedade, sem modificar nada.
* **Setter**: É um método que **define** (ou "seta") o valor de uma propriedade. O termo vem do verbo "to set", que significa "definir" ou "estabelecer". O setter é usado para atribuir um novo valor a uma propriedade, e pode incluir validações para garantir que o valor fornecido seja aceitável.
* **Getter**: Apenas **retorna** o valor da propriedade, sem modificá-lo.
* Texto

  Descrição gerada automaticamente**Setter**: **Define** um novo valor para a propriedade, podendo incluir validações.

Acessamos o private através da atribuição dele com o recebimento do parâmetro em setter, armazenamos o valor e depois retornamos com um getter.

O objetivo principal do método **setter** é **definir** o valor de uma propriedade, e não retornar dados. Geralmente, o **setter** é projetado para receber um valor, realizar a validação necessária e, então, armazenar o valor na propriedade da classe, enquanto o **getter** é responsável por **retornar** esse valor.

Colocar um return dentro de um setter pode ser confuso, pois quebra a convenção de design, onde o setter deve apenas definir um valor e o getter deve apenas retornar o valor. No entanto, é possível retornar algo de um setter, embora isso seja raro e não recomendado na maioria das práticas de programação orientada a objetos.

### Visibilidade de Propriedades Assimétricas

No contexto de visibilidade de propriedades no PHP (a partir da versão 8.4), **assimetria** refere-se à capacidade de definir diferentes níveis de acesso para **leitura** e **escrita** de uma mesma propriedade. Em outras palavras, a visibilidade assimétrica permite que uma propriedade tenha permissões distintas para **obter** (ou "ler") seu valor e para **definir** (ou "escrever") um novo valor.

Essa possibilidade foi implementada apenas no PHP 8.4

A palavra "assimetria" vem do prefixo "a-", que significa "não", e "simetria", que implica igualdade ou equilíbrio. Portanto, **assimetria** significa "falta de simetria" ou "desigualdade". No caso de visibilidade assimétrica, isso implica que o acesso para ler e escrever uma propriedade não é "simétrico" ou "igual". Cada operação (leitura e escrita) pode ter sua própria visibilidade e não precisam ter os mesmos níveis de acesso.

*Existem algumas regras para isso:*

* **Visibilidade Restrita de Escrita**: A visibilidade de escrita (set) deve ser igual ou mais restritiva que a visibilidade de leitura (get).
* **Propriedades private(set) são final**: Quando uma propriedade é declarada como private(set), ela é automaticamente final, impedindo que subclasses sobrescrevam a propriedade.
* **Somente Propriedades Tipadas**: Apenas propriedades com tipos definidos podem ter visibilidade assimétrica.

*Combinações:*

1. public public(set) string $name;
   1. Leitura e escrita são públicas. Qualquer parte do código pode ler e modificar a propriedade.
2. public protected(set) string $name;
   1. Leitura é pública, mas a escrita é protegida. Qualquer parte do código pode ler o valor da propriedade, mas apenas a própria classe e suas subclasses podem modificá-la.
3. public private (set) string $name;
   1. Leitura é pública, mas a escrita é privada. Qualquer parte do código pode ler o valor da propriedade, mas apenas a própria classe pode modificá-la.
4. protected protected(set):
   1. Leitura e escrita são protegidas. Somente a própria classe e suas subclasses podem ler e modificar a propriedade.
5. protected private(set):
   1. Leitura é protegida, mas a escrita é privada. A propriedade pode ser lida pela própria classe e suas subclasses, mas apenas a própria classe pode modificá-la.
6. private private(set):
   1. Leitura e escrita são privadas. Somente a própria classe pode ler e modificar a propriedade.

O sufixo (set) na visibilidade assimétrica foi introduzido no PHP 8.4 para indicar especificamente o nível de **visibilidade de escrita** (ou "set") de uma propriedade, diferenciando-a da visibilidade de **leitura** (ou "get").

# Herança de Objetos

### Extends

Uma classe pode herdar métodos e propriedades de outra classe usando a palavra-chave extends na declaração da classe. Não é possível herdar múltiplas classes; uma classe só pode herdar uma classe base.

Quando você cria uma classe que "estende" outra, a nova classe (chamada **classe filha** ou **subclasse**) da classe original (chamada **classe pai** ou **superclasse**), exceto aqueles que são declarados como private na classe pai.

Texto

Descrição gerada automaticamenteIsso é útil para definir métodos adicionais e opcionais sem a necessidade de repetir código.

# Construtores e Destrutores

Com vimos anteriormente sobre construtores e destrutores, aqui iremos apenas dar continuidade a algumas informações adicionais.

Quando uma classe filha, herda do pai uma construtor, o construtor do pai é herdado normalmente, mas se a classe filha também tiver um construtor, é necessário utilizar a chamada “parent::\_\_construtc” dentro da classe filha. Caso o construtor do pai seja privado, não irá ser herdado.

Texto

Descrição gerada automaticamente

### Promoção de Propriedades no Construtor

Quando vamos utilizar o construtor, primeiro declaramos as propriedades que serão usadas, depois construimos o nosso construtor e ligamos os parâmetros as propriedades, da seguinte forma:

Texto

Descrição gerada automaticamente

Nesse exemplo, a propriedade $nome e $idade são declaradas primeiro e depois inicializadas no construtor. Esse código funciona bem, mas é repetitivo.

Texto

Descrição gerada automaticamenteA promoção de propriedades permite fazer tudo isso **diretamente no construtor**, eliminando a necessidade de declarar e inicializar as propriedades separadamente. Com essa nova funcionalidade, o código acima pode ser reescrito assim:

Ou seja, sempre que colocamos uma visibilidade ou outro modificador no parâmetro do construtor, o PHP automaticamente são declaradas e iniciadas dentro do construtor.

**Como Funciona**

* Quando você escreve public string $nome dentro dos parênteses do construtor, você está dizendo ao PHP para **criar uma propriedade pública chamada $nome do tipo string** e **definir o valor dela** com o valor passado no construtor.
* O mesmo vale para outras propriedades que você incluir no construtor.
* No PHP, **propriedades de objeto não podem ser do tipo callable** devido à ambiguidade que isso poderia introduzir.

# Operador de Resolução de Escopo (::)

### Membros estáticos

Em PHP, **membros estáticos** são propriedades e métodos de uma classe que podem ser acessados **sem a necessidade de instanciar o objeto**. Eles são definidos com a palavra-chave static e são compartilhados entre todas as instâncias da classe, ou seja, pertencem à própria classe, e não a objetos individuais criados a partir dela.

**Acesso Direto**: Você pode acessar membros estáticos diretamente pela classe, usando

NomeDaClasse : : membroEstatico

Usamos o operador “::” para fazer isso.

Texto

Descrição gerada automaticamenteO **operador ::** (conhecido como **operador de resolução de escopo** ou **operador de escopo de classe**) é usado para acessar membros estáticos, métodos ou constantes diretamente pela classe, sem a necessidade de instanciar um objeto.

**Diferença entre $this e self::**

* **$this**: É usado para acessar membros (propriedades e métodos) de uma instância específica da classe. Ou seja, ele é usado dentro de métodos não estáticos, onde $this representa o próprio objeto.
* **self::**: É usado dentro de métodos estáticos para acessar membros estáticos. Como métodos estáticos não têm uma instância associada, $this não está disponível, então self:: é usado para acessar membros estáticos da classe.

Os membros estáticos em PHP **não são constantes**; eles podem ser **modificados e processados**. A diferença principal é que eles pertencem à classe em si e são **compartilhados entre todas as instâncias** da classe, mas isso não significa que sejam imutáveis.

O conceito de **membros estáticos** pode parecer um pouco contraintuitivo, já que, de fato, o propósito principal de uma classe é servir como um **molde para instanciar objetos**. No entanto, os membros estáticos têm usos específicos que são úteis em certos cenários. Vou explicar quando e por que eles são usados.

**Exemplos de uso de membros estáticos:**

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Os membros estáticos são úteis quando você precisa de **dados ou funcionalidades que pertencem à classe como um todo**, e não a uma instância específica. Em resumo, você usaria membros estáticos para:

* **Armazenar valores globais ou configuráveis** que devem ser compartilhados entre todas as instâncias.
* **Manter contadores ou estatísticas** relacionadas ao uso da classe.
* **Definir métodos utilitários** que não dependem de instâncias.
* **Implementar certos padrões de projeto** onde o estado ou o comportamento deve ser compartilhado globalmente.

# Palavra-Chave 'static'

Quando estamos trabalhando com programação procedural, é organizo em arquivos diferente as funções que trabalham criando instância de dados das funções que servem como “constantes” apenas para processar algum tipo de dados para realizar um cáculo ou outra coisa “estática”.

Texto

Descrição gerada automaticamenteMétodos estáticos são ótimos para **funções que realizam uma tarefa específica ou processam dados**, mas que **não dependem de nenhum dado interno do objeto** (ou seja, não precisam de nenhuma propriedade específica de uma instância). Eles apenas recebem os valores necessários como parâmetros, fazem o trabalho, e retornam o resultado.

Você pode pensar nos **métodos estáticos** como algo **parecido com uma constante** em termos de comportamento.

Logotipo

Descrição gerada automaticamente com confiança baixaE dentro dessas classes que agrupam métodos estáticos, como eles não criam nenhum tipo de objeto e apenas processam dados, a forma como podemos acessar esses métodos e propriedades estáticas é com operador duplo “::” como vimos anteriormente.

Mas diferente dos métodos estáticos, temos também as propriedades estáticas.

E as propriedades estáticas, como não estão dentro de nenhum método, elas mesmas pertencem a classe, e como já disse, como elas são estáticas e NÃO instânciam um novo objeto, elas basicamente podem ser modificadas de qualquer lugar onde forem chamadas.

# Abstração de Classes

A abstração em PHP permite que você defina classes e métodos que servem como "modelos" para outras classes, sem precisar implementar detalhes específicos.

Esse “modelos” a serem seguidos são obrigatórios e sempre devem ser extendidos pelas subclasses.

Para isso utilizamos a palavra chave “abstract”

**Ex:**

Texto

Descrição gerada automaticamenteEu crio uma class que vai ser apenas o modelo:

O que esse código acima quer dizer, é algo parecido com:

“Esta classe não pode ser instanciada diretamente, estenda ela para um classe filha e crie obrigatoriamente um método publico chamado makeSound, para pode usar a classe e o método”.

Texto

Descrição gerada automaticamente

**Conceitos Básicos**

1. **Classe Abstrata**: É uma classe que não pode ser instanciada diretamente. Ela serve como uma base para outras classes.
2. **Método Abstrato**: É um método que não tem implementação na classe abstrata; ele deve ser implementado nas classes filhas.

**Como Funciona**

* Uma **classe abstrata** é declarada com a palavra-chave abstract. Ela pode ter métodos normais (com corpo) e métodos abstratos (sem corpo).
* Um **método abstrato** é apenas declarado, e as classes que herdam essa classe precisam terminar de construir esse método.

**Mas afinal, pra que serve ?**

1. **Organização do Código**: A abstração permite que você crie "modelos" de classes que definem apenas o que é essencial, sem os detalhes. Isso deixa o código mais organizado e fácil de entender.
2. **Regras Consistentes**: Com métodos abstratos, você força as classes filhas a implementarem certos métodos. Por exemplo, se todos os animais devem "fazer som", a classe abstrata Animal garante que todos os animais (como Dog, Cat) vão ter o método makeSound(), mas cada um com sua própria implementação.
3. **Flexibilidade e Expansão**: A abstração permite criar uma estrutura que pode ser facilmente expandida. Se você tem uma classe Animal, pode adicionar quantos tipos de animais quiser (cachorro, gato, etc.), e cada um implementa seus próprios detalhes, mas todos seguem a estrutura de Animal.

**Exemplo Real**

Imagine que você tem um sistema de pagamento. Uma classe abstrata Pagamento pode definir o que todos os tipos de pagamento precisam ter, como um método processar(), mas deixa para cada classe filha (CartaoCredito, Boleto, etc.) implementar como o processamento ocorre. Assim, você tem certeza de que todo método de pagamento terá processar(), mas cada um será feito de forma específica.

# Interfaces de Objetos

A interface é parecida com abstract, mas com algumas diferenças.

Tanto a classe abstrata quanto a interface "obrigam" as classes que as utilizam a implementar certos métodos, mas com propósitos e funcionalidades finais diferentes.

**Quando usar cada uma:**

* Use **classe abstrata** quando há uma relação de herança lógica (exemplo: Animal > Dog).
* Use **interface** quando você quer garantir que várias classes diferentes implementem os mesmos métodos, sem se preocupar com a implementação em si (exemplo: Movable para qualquer coisa que possa se mover).

Texto

Descrição gerada automaticamenteE para isso, utilizamos a palavra chave interface no lugar de class:

**Resumo das Diferenças**

|  |  |  |
| --- | --- | --- |
| Característica | Classe Abstrata | Interface |
| |  | | --- | | Implementação de métodos |  |  | | --- | |  | | Pode ter métodos concretos | Somente métodos abstratos |
| Atributos | Pode ter atributos | Não pode ter atributos |
| Herança | Apenas uma por classe | Uma classe pode implementar várias |
| Propósito | Base para outras classes | Contrato para implementar métodos |

Texto

Descrição gerada automaticamente

Todos os métodos declarados em uma interface devem ser públicos, essa é a natureza de uma interface.

Constantes dentro de interfaces podem ser sobrescritas quando puxadas para as classes.

# Traits

Quando o código fica muito grande, e estamos com diversas classes herdando outras muitas classes, utilizamos as “traits” com a palavra chave “use” e “trait”.

**Conceitos Básicos de Traits**

1. **Reutilização de Código**: Traits permitem que você escreva funcionalidades em um só lugar e as "injete" em várias classes, sem que essas classes precisem ser relacionadas por herança.
2. **Combinação com Outras Traits e Métodos**: Você pode combinar várias traits em uma mesma classe e, se houver conflito de métodos (mesmo nome de método em duas traits), você pode resolver especificando qual método usar.
3. **Sintaxe Simples**: Traits são definidas com a palavra-chave trait e usadas nas classes com use, e não precisam extender outras classes, apenas precisam estar no mesmo arquivo ou ligadas com “inlcude/require” e utilizar dentro das classes.

Texto

Descrição gerada automaticamente

**Por Que Usar Traits?**

As traits são úteis quando você precisa compartilhar métodos entre classes que não podem (ou não devem) estar na mesma hierarquia. Por exemplo, User e Product não têm relação de herança, mas ambos precisam de um método log(). Usar uma trait ajuda a manter o código limpo e evitar repetição.

Pq afinal, na orientação a objetos quando fazemos a análise de requisitos, identificamos objetos e descobrimos e definimos métodos, vai existir métodos que vão se repetir em muitos objetos sem fazer parte da estrutura daquele objeto. Portanto, podemos separa-los em uma espécie de classe única, com intuito de poder implementa-lo sem a necessidade de hierarquizar alguma coisa.

# Classes anônimas

Classes anônimas são basicamente classes recebidas em variáveis.

Elas são úteis para criar objetos "descartáveis" que não precisam ser reutilizados, oferecendo uma forma rápida de criar objetos simples sem definir uma classe completa.

**Características das Classes Anônimas**

1. **Sem Nome**: São criadas sem nome, o que as torna ideais para uso único, quando você só precisa de um objeto temporário.
2. **Definidas Inline**: Você pode definir uma classe anônima diretamente dentro do código, o que economiza espaço e evita a criação de uma nova classe no escopo global.
3. **Podem Implementar Interfaces e Extender Classes**: Embora sejam anônimas, elas podem implementar interfaces e estender outras classes, o que as torna bastante flexíveis.

**Vantagens de Usar Classes Anônimas**

1. **Código Mais Limpo**: Evitam a criação de classes específicas para funcionalidades pequenas e temporárias.
2. **Encapsulamento Rápido**: Permitem criar objetos com lógica específica em tempo real, mantendo o código modular.
3. Texto

   Descrição gerada automaticamente**Flexibilidade**: Podem implementar interfaces e herdar classes, mantendo as vantagens da orientação a objetos.

**Quando Usar Classes Anônimas?**

Classes anônimas são ideais para situações onde você precisa de um objeto simples e temporário, especialmente em testes, callbacks, ou para passar comportamentos específicos a funções sem criar uma classe completa. Elas ajudam a reduzir a quantidade de classes definidas e a manter o código organizado.

Quando trabalhamos com orientação a objetos (OO), a ideia é evitar funções soltas no código (estilo procedural) para manter a **organização e o encapsulamento** que o paradigma OO oferece. Classes anônimas são uma excelente alternativa para criar comportamentos temporários ou específicos **sem recorrer a funções soltas** e mantendo o estilo OO.

# Métodos mágicos

Métodos mágicos são métodos especiais que sobrescrevem o comportamento padrão do PHP quando certas operações são realizadas em um objeto.

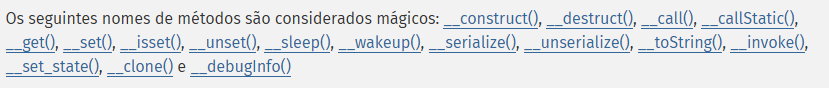
Todos os métodos prefixados com ”\_\_”  são reservados pelo PHP. Portanto, não é recomendado utilizar nomes de métodos com esse prefixo a não ser para sobrescrever o comportamento do PHP.

Os métodos mágicos em PHP usam o prefixo \_\_ (dois sublinhados) para diferenciá-los dos métodos comuns e indicar que são especiais apenas.

**Quando Usar Métodos Mágicos?**

Métodos mágicos são poderosos para personalizar comportamentos e facilitar a manutenção de código dinâmico, mas seu uso excessivo pode dificultar o entendimento do código. Devem ser usados com moderação, especialmente \_\_get e \_\_set, que podem tornar o código mais complexo de depurar.

Os métodos mágicos \_\_construct, \_\_destruct & \_\_clone precisam ser declarados como public, senão um erro warning será lançado.

Temos os seguintes métodos mágicos:

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Parece contraintuitivo criar métodos para lidar com propriedades ou métodos "inexistentes". No entanto, há várias razões práticas para isso em sistemas complexos, onde a flexibilidade e a escalabilidade são essenciais. Vamos ver por que isso pode ser útil e como ele realmente **completa o código** ao invés de deixá-lo "incompleto".

*Podemos separa os métodos mágicos nos seguintes grupos:*

**1. Inicialização e Finalização do Objeto**

* **\_\_construct()**: Chamado quando o objeto é criado, usado para inicializar o estado do objeto.
* **\_\_destruct()**: Chamado quando o objeto é destruído, usado para liberar recursos.

**2. Manipulação de Propriedades**

* **\_\_get($name)**: Chamado ao tentar acessar uma propriedade inexistente ou inacessível.
* **\_\_set($name, $value)**: Chamado ao tentar definir uma propriedade inexistente ou inacessível.
* **\_\_isset($name)**: Chamado ao usar isset em uma propriedade inexistente ou inacessível.
* **\_\_unset($name)**: Chamado ao usar unset em uma propriedade inexistente ou inacessível.

**3. Sobrecarga de Métodos**

* **\_\_call($name, $arguments)**: Chamado ao invocar um método inexistente na instância do objeto.
* **\_\_callStatic($name, $arguments)**: Chamado ao invocar um método inexistente de forma estática.

**4. Serialização e Desserialização**

* **\_\_sleep()**: Chamado antes de serializar o objeto, define as propriedades a serem salvas.
* **\_\_wakeup()**: Chamado ao desserializar o objeto, usado para restaurar o estado.
* **\_\_serialize()**: Define os dados a serem serializados (PHP 7.4+).
* **\_\_unserialize()**: Reconstrói o objeto a partir dos dados desserializados (PHP 7.4+).

**5. Representação do Objeto como String ou Função**

* **\_\_toString()**: Chamado ao tratar o objeto como string (ex., echo $obj;), retorna uma representação textual.
* **\_\_invoke()**: Chamado ao tratar o objeto como uma função (ex., $obj();), tornando-o "invocável".

**6. Manipulação Especial e Outros Casos Específicos**

* **\_\_set\_state($array)**: Chamado ao exportar uma classe com var\_export(), reconstruindo o objeto a partir de um array.
* **\_\_clone()**: Chamado ao clonar o objeto com clone, permitindo copiar dados específicos.
* **\_\_debugInfo()**: Define quais informações são mostradas ao usar var\_dump() no objeto, controlando a depuração.

**Flexibilidade para Dados Variáveis**

Em algumas aplicações, você não sabe com antecedência todos os atributos que um objeto precisará ter. Um exemplo comum são sistemas de **e-commerce** ou **gestão de dados** onde os atributos de produtos, usuários, ou configurações podem variar.

Imagine:

* Um sistema de produtos que precisa armazenar atributos específicos para cada tipo de produto (como cor, tamanho para roupas, mas capacidade e peso para eletrônicos).
* Ao usar métodos mágicos e sobrecarga, você evita declarar antecipadamente todas as propriedades possíveis, permitindo armazenar dados de forma dinâmica e limpa.

Esse recurso **não deixa o código incompleto** — ele o torna **mais adaptável** a diferentes cenários sem precisar modificar a estrutura da classe a cada nova necessidade.

**Reduz Duplicação e Aumenta a Manutenção**

Se você tentasse declarar todas as propriedades e métodos possíveis para cada cenário, isso:

* **Incharia o código** com declarações desnecessárias.
* Tornaria o código mais difícil de manter, já que a cada novo atributo ou método você teria que modificar a classe.

Usar sobrecarga e métodos mágicos centraliza o controle de acesso e manipulação de propriedades, tornando o código mais enxuto e fácil de atualizar.

**Facilita a Criação de APIs e Objetos Dinâmicos**

Muitos frameworks modernos, como o Laravel, usam sobrecarga para facilitar a criação de APIs dinâmicas, onde os atributos e métodos podem variar conforme o contexto. Por exemplo:

* Um sistema de **ORM** (Mapeamento Objeto-Relacional) que permite acessar dinamicamente colunas de um banco de dados como propriedades de um objeto. Esse tipo de flexibilidade é fundamental para que o código funcione com diferentes esquemas de dados sem precisar de alterações constantes.

# Sobrecarga

No contexto do PHP, **sobrecarga** não funciona como em outras linguagens orientadas a objetos (onde você pode definir vários métodos com o mesmo nome, mas diferentes parâmetros). Em PHP, a sobrecarga se refere ao **uso dinâmico de métodos e propriedades que não foram declarados na classe**. Ou seja, conseguir tratar de dados sem necessariamente estarem em uma propriedade ou método construído dentro da classe.

**Por Que É Chamado de "Sobrecarga"?**

O nome "sobrecarga" em PHP vem da ideia de "carregar sobre" ou "acrescentar" um comportamento extra, permitindo o acesso dinâmico a propriedades e métodos sem que eles estejam declarados diretamente na classe.

**2. Sobrecarga de Propriedades**

Sobrecarga de propriedades permite capturar acessos a propriedades que não foram declaradas explicitamente na classe, usando métodos mágicos como:

* \_\_set($name, $value): Invocado ao definir uma propriedade inexistente.
* \_\_get($name): Invocado ao acessar uma propriedade inexistente.
* \_\_isset($name): Invocado ao verificar a existência de uma propriedade.
* \_\_unset($name): Invocado ao remover uma propriedade inexistente.

Texto

Descrição gerada automaticamente

**3. Sobrecarga de Métodos**

A sobrecarga de métodos em PHP permite que você capture chamadas para métodos "inexistentes" usando os métodos mágicos:

* \_\_call($name, $arguments): Invocado ao chamar um método inexistente.
* \_\_callStatic($name, $arguments): Invocado ao chamar um método inexistente estaticamente.

Texto

Descrição gerada automaticamenteEsses métodos capturam a chamada e permitem que você gerencie as invocações de maneira dinâmica.

**Quando Usar Sobrecarga?**

* **Acessos Dinâmicos**: Útil em situações onde você precisa de acessos dinâmicos a propriedades ou métodos, como em APIs ou objetos que encapsulam dados flexíveis.
* **Encapsulamento de Dados**: Pode ajudar a encapsular dados sem definir todas as propriedades e métodos, especialmente em frameworks e aplicações complexas.

# Iteração de Objetos

A **iteração de objetos** em PHP permite que você percorra as propriedades de um objeto como se fosse um array, usando estruturas como foreach. Essa funcionalidade é útil para acessar ou manipular dados de um objeto sem precisar saber antecipadamente quais propriedades ele possui.

**Quando Usar a Iteração de Objetos?**

1. **Acessar Dados em Massa**: Em objetos que contêm muitas propriedades, especialmente em coleções de dados ou resultados de banco de dados.
2. **Controlar a Exposição de Dados**: Com Iterator ou IteratorAggregate, você controla exatamente quais dados são expostos e como, ideal para proteger informações sensíveis.
3. Texto

   Descrição gerada automaticamente**Customizar a Navegação de Objetos**: Você pode definir uma ordem ou até mesmo transformar os dados enquanto os itera.

# Final

Em PHP, a palavra-chave final é usada para impedir que classes ou métodos sejam estendidos ou sobrescritos. Ela serve para garantir que certas partes do código permaneçam imutáveis, ajudando a proteger a lógica e a integridade de uma classe ou método.

**1. final em Classes**

Texto

Descrição gerada automaticamenteQuando uma classe é declarada como final, ela **não pode ser herdada**. Isso significa que nenhuma outra classe pode estender essa classe final. Isso é útil quando você quer garantir que a implementação da classe permaneça exatamente como está, sem modificação por subclasses.

**2. final em Métodos**

Quando um método é declarado como final, ele **não pode ser sobrescrito** em subclasses. Você pode herdar a classe, mas o método final permanecerá o mesmo em todas as subclasses, mantendo sua implementação original.

Texto

Descrição gerada automaticamente

**Quando Usar final?**

1. **Proteger a Implementação**: final é útil para classes e métodos que não devem ser alterados. Isso ajuda a proteger a lógica interna, garantindo que não seja modificada em subclasses.
2. **Segurança em APIs e Bibliotecas**: Em bibliotecas ou APIs que outras pessoas podem usar, final impede que usuários modifiquem métodos críticos, ajudando a prevenir comportamentos inesperados.
3. **Facilitar a Manutenção**: Ao garantir que partes do código não serão alteradas, final ajuda a manter a consistência do sistema, facilitando a manutenção.

# Clonando objetos

A clonagem de objetos em PHP é útil em situações em que você precisa de uma **cópia independente** de um objeto, onde alterações feitas na cópia não afetem o objeto original. Isso é importante em diversos contextos de desenvolvimento, onde o estado dos objetos precisa ser preservado ou manipulado separadamente.

Em situações onde você precisa experimentar ou simular mudanças em um objeto sem alterar o original, a clonagem é útil. Isso é feito usando a palavra-chave **clone**, que cria um novo objeto com os mesmos valores do objeto original.

**Clonagem Simples**

Quando você usa clone, o PHP cria uma cópia "superficial" do objeto. Isso significa que o PHP copia todos os valores das propriedades, mas **não cria cópias dos objetos aninhados** (caso existam referências a outros objetos). Em vez disso, ele mantém as referências, o que pode levar a comportamentos inesperados.

Texto

Descrição gerada automaticamente

**Tipos de Clonagem**

1. **Clonagem Superficial**: Cria cópias das propriedades, mas **mantém as referências de objetos internos**.
2. **Clonagem Profunda**: Usa \_\_clone() para criar cópias reais dos objetos internos, garantindo que as alterações no clone não afetem o objeto original.

**O Método Mágico \_\_clone()**

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamenteQuando você clona um objeto, o PHP permite que você personalize o processo de clonagem usando o método mágico \_\_clone(). Esse método é chamado automaticamente após a criação do clone e pode ser usado para ajustar propriedades ou até mesmo clonar objetos internos para criar uma cópia "profunda".



# Comparando objetos

A **comparação de objetos** em PHP permite verificar se dois objetos são iguais ou idênticos. Em PHP, há duas maneiras principais de comparar objetos: **comparação de igualdade (==)** e **comparação de identidade (===)**. Cada uma delas possui características e é útil para diferentes cenários.

**1. Comparação de Igualdade (==)**

Ao usar == para comparar objetos, o PHP verifica se:

* Os objetos são instâncias da mesma classe.
* As propriedades dos objetos têm os mesmos valores.

Esta comparação ignora o fato de os objetos serem instâncias distintas, focando apenas nos valores de suas propriedades.

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

**2. Comparação de Identidade (===)**

Ao usar ===, o PHP verifica se:

* Os objetos são instâncias da mesma classe.
* As propriedades dos objetos têm os mesmos valores.
* Ambos os objetos referenciam a **mesma instância** (ou seja, ocupam o mesmo espaço na memória).

Texto

Descrição gerada automaticamenteA comparação === verifica se os dois objetos são exatamente o mesmo objeto.

**Implementando Comparação Personalizada com Comparable**

Se você precisar de uma lógica personalizada para comparar objetos, pode implementar uma interface ou criar um método específico, como equals(). Isso é útil quando você quer mais controle sobre o que significa "igualdade" para os seus objetos.

Texto

Descrição gerada automaticamente

O método equals compara explicitamente o nome e o preço dos produtos. Isso permite uma comparação focada nos atributos específicos que você deseja considerar para igualdade.

# Late Static Bindings

Texto

Descrição gerada automaticamenteQuando temos diversas classes herdando uma a outra, acabamos as vezes por criar uma hierarquia de classes.

**Hierarquia de Classes**: Quando criamos classes que herdam umas das outras, formamos uma hierarquia, onde as classes "filhas" herdam métodos e propriedades das classes "pais". Isso permite reutilizar e estender o comportamento de classes existentes.

* **Uso de self e this**:
* **self**: Quando você usa self em um método estático, ele sempre se refere à classe em que o método foi definido, mesmo que a classe seja herdada. Isso significa que o comportamento será o da classe base, onde o método foi originalmente criado.
* **this**: É usado para acessar propriedades e métodos dentro da instância de uma classe. Funciona no nível de instância e se refere ao objeto que chamou o método.
* **Usando static**:
* Quando queremos que o método use a classe que o está chamando (e não a classe onde o método foi originalmente definido), usamos static::. O static refere-se à classe que fez a chamada, mesmo que o método esteja definido em uma classe pai.
* Isso permite que métodos em uma classe base se comportem de forma dinâmica, adaptando-se às subclasses que os estendem.

Texto

Descrição gerada automaticamente

**OBS:** A palavra static foi escolhida pelos desenvolvedores como palavra reservada para essa funcionalidade simplesmente pq ela já estava disponível para tal.

Foi decidido não introduzir uma nova palavra-chave, mas usar static, já reservada.

**Atenção!!**  
Isso só pode ser utilizado com métodos estáticos!

# Objetos e Referências

**Objetos em PHP Não São Passados por Referência, Mas Sim por Identificadores**

* Quando você cria um objeto em PHP, a variável que contém o objeto não armazena o objeto em si. Em vez disso, ela armazena um **identificador** que aponta para o objeto na memória.
* Quando você atribui um objeto a outra variável, você não cria uma cópia do objeto. Em vez disso, ambas as variáveis compartilham o mesmo identificador, que aponta para o mesmo objeto.

Texto

Descrição gerada automaticamente

Quando alteramos $b->foo para 2, $a->foo também é 2, porque $a e $b se referem ao mesmo objeto.

# Serialização de Objetos

A **serialização de objetos** em PHP é um recurso que transforma um objeto em uma string que representa o estado desse objeto. Isso é útil quando você deseja armazenar ou transferir objetos, por exemplo, para salvar dados em uma sessão, em um arquivo, ou para enviar informações através de uma rede.

Imagine que você tem um objeto com vários dados importantes e deseja:

1. **Armazenar o objeto** para usá-lo mais tarde (por exemplo, em um arquivo ou no banco de dados).
2. **Enviar o objeto** para outra página ou para outro sistema (por exemplo, usando sessões ou transferindo dados via rede).

A serialização permite fazer isso de forma eficiente, transformando o objeto em uma string que pode ser facilmente armazenada ou enviada.

Texto

Descrição gerada automaticamentePense em uma caixa com vários itens (o objeto). Você quer guardar essa caixa em um local pequeno ou enviar por correio (salvar ou transferir). Para fazer isso, você tira uma foto detalhada da caixa (a string serializada), guarda ou envia a foto, e mais tarde pode recriar a caixa com todos os itens usando a foto (desserializar).

**serialize()**: Converte o objeto $usuario em uma string que contém todas as informações sobre as propriedades do objeto. Você pode armazenar ou transferir essa string.

**unserialize()**: Recria o objeto original a partir da string serializada. O objeto $usuarioRecriado tem as mesmas propriedades e valores que o objeto $usuario.

**O Que é Salvo na Serialização?**

* **Propriedades do Objeto**: A serialização salva os valores das propriedades (dados) do objeto.
* **Nome da Classe**: A serialização também salva o nome da classe para saber como recriar o objeto.
* **Métodos Não São Salvos**: Os métodos (funcionalidades) do objeto não são armazenados. Para usar o objeto desserializado corretamente, a definição da classe deve estar presente no código.

**Por Que Isso é Útil?**

1. **Salvar o Estado do Objeto**: Se você quiser salvar o estado atual de um objeto (por exemplo, um carrinho de compras) para usá-lo mais tarde ou após um reinício do sistema.
2. **Armazenamento em Sessões**: Em aplicações web, você pode querer manter o estado de um usuário entre as páginas. Serializar o objeto permite armazená-lo na sessão.
3. **Transferência de Dados**: Se você precisar enviar dados complexos para outro sistema ou processar dados em um ambiente diferente, a serialização transforma o objeto em um formato que pode ser transferido e restaurado.

# Covariância e Contravariância

**Covariância**: Imagine que você tem um método que deveria "entregar" algo, como um Animal. Com covariância, você permite que a classe filha entregue algo **mais específico** que um Animal, como um Cachorro ou um Gato.

**Contravariância**: Agora imagine que você tem um método que aceita algo como comida para animais. Com contravariância, você permite que a classe filha aceite algo **menos específico**, como qualquer tipo de comida, e não apenas comida para animais.

Mas antes de vermos exemplos, primeiro precisamos como é que isso foi adicionado ao PHP.

Antes do PHP 7.4, o sistema de tipos em PHP era mais limitado. Se você quisesse sobrescrever um método em uma classe filha, os tipos de parâmetros e de retorno precisavam seguir regras mais estritas, sem a flexibilidade de permitir tipos mais específicos ou mais genéricos.

Por exemplo:

* Você não podia fazer um método em uma classe filha retornar um tipo mais específico que o método da classe pai.
* Da mesma forma, os parâmetros de um método em uma classe filha não podiam ser menos específicos do que na classe pai.

O conceito de **covariância e contravariância** em PHP aplica-se principalmente a **tipos de retorno e parâmetros que envolvem objetos como retorno**. Quando falamos de tipos de dados padrão, como string, int, ou bool, as regras de tipagem são mais simples e não há a mesma necessidade de flexibilidade.

**Tipos Objetos**: Covariância e contravariância são relevantes quando lidamos com classes e hierarquias de objetos. Isso ocorre porque as classes podem ter relacionamentos de herança

* (ex.: uma classe Dog que herda de Animal), e há a necessidade de controlar a especificidade dos tipos de retorno e a generalidade dos parâmetros.

**Tipos Primitivos**: Para tipos padrão como string, int, ou float, a questão de covariância e contravariância não se aplica da mesma forma.

* Se um método em uma classe pai retorna um string, a classe filha também deve retornar um string. Não há a flexibilidade de especificidade ou generalidade porque os tipos primitivos não são objetos que podem ter subclasses ou superclasses.

**Exemplo de Covariância**

Permite que métodos em **classes filhas** tenham um **tipo de retorno mais específico** que o tipo de retorno definido no método da **classe pai**. Ou seja, em vez de retornar um objeto mais genérico, como Animal, o método na classe filha pode retornar um objeto mais específico, como Dog.

* **Exemplo Simples de Covariância**: Se o método na classe pai retorna um Animal, o método na classe filha pode retornar um Dog ou Cat, que são subclasses de Animal. Isso permite que o método na classe filha seja mais específico sobre o que ele retorna.

Texto

Descrição gerada automaticamente

* A interface AnimalShelter diz que qualquer abrigo deve adotar um **Animal**.
* DogShelter é um abrigo que especifica que vai adotar um **Cachorro**, que são tipos mais específicos de **Animal**, porém podem ser retornados justamente por serem classes filhas.
* Isso é **covariância**: permitir que uma classe filha seja retornada, assim sendo mais específico ao definir o tipo de retorno para a classe pai.

**Covariância =** Tipo de retorno de objeto mais específico.

**Exemplo de Cotravariância**

Permite que métodos em **classes filhas** aceitem **parâmetros mais gerais** ou menos específicos que o tipo definido no método da **classe pai**. Em vez de exigir um tipo específico de parâmetro, o método na classe filha pode aceitar algo mais amplo.

* Texto

  Descrição gerada automaticamente**Exemplo Simples de Contravariância**: Se o método na classe pai aceita um AnimalFood, o método na classe filha pode aceitar um Food, que é mais genérico. Isso torna o método na classe filha mais flexível sobre o que pode ser passado como argumento.
* A classe Animal só aceita **AnimalFood**. Mas a classe Dog é mais flexível e aceita **qualquer tipo de comida**.
* Isso é **contravariância**: permitir que a classe filha aceite algo menos específico.

# Objetos lentos

**Objetos lentos** são objetos cuja **inicialização é adiada** até que seu estado seja observado ou modificado. Em outras palavras, eles não são totalmente carregados ou inicializados quando são criados. Em vez disso, eles só são inicializados quando você realmente precisar usar alguma de suas propriedades ou métodos.

Esse conceito é muito útil para melhorar a **performance** de uma aplicação, especialmente em situações onde inicializar um objeto pode ser caro (por exemplo, acessar um banco de dados, carregar grandes quantidades de dados, ou realizar cálculos complexos).

**Exemplos de Usos de Objetos Lentos**

1. **Injeção de Dependência**: Quando um serviço ou componente é caro de inicializar, ele só é criado quando realmente for necessário.
2. **ORMs (Mapeamento Objeto-Relacional)**: Entidades do banco de dados podem ser carregadas apenas quando você acessa suas propriedades.
3. **Analisadores JSON ou XML**: O parser pode adiar a análise completa dos dados até que você precise acessar elementos específicos.

**Estratégias de Inicialização**

Uma **estratégia** é um conjunto de técnicas ou métodos que você escolhe com base no que funciona melhor para o seu caso específico. Quando falamos de objetos lentos em PHP, temos duas principais estratégias para implementar o conceito de inicialização tardia:

1. **Objetos Fantasmas (Lazy Ghosts)**
2. **Proxies Virtuais (Lazy Proxies)**

Você escolhe a estratégia que melhor se adapta às necessidades da sua aplicação. Vamos entender melhor o que cada uma faz e por que você poderia escolher uma em vez da outra.

1. **Estratégia de Objetos Fantasmas (Lazy Ghosts)**

* **O que é**: Um objeto fantasma é um objeto que parece estar completamente inicializado, mas na verdade só será inicializado quando você realmente precisar acessar ou modificar suas propriedades.
* **Como Funciona**: Quando você tenta acessar uma propriedade pela primeira vez, o objeto é inicializado de forma transparente.
* **Quando Usar**: Essa estratégia é útil quando você controla totalmente como e quando o objeto será inicializado. Por exemplo, em um sistema onde você precisa ter certeza de que o objeto se comporta como um objeto "normal" após a inicialização.

**Por que é uma estratégia?**: Porque é uma abordagem que prioriza a simplicidade e transparência. Você planeja usar essa estratégia quando precisa que os objetos se comportem de maneira previsível, sem que o código que usa o objeto precise saber que ele é "preguiçoso".

1. **Estratégia de Proxies Virtuais (Lazy Proxies)**

* **O que é**: Um proxy virtual é um "substituto" que atua em nome do objeto real. O objeto real só é criado quando você realmente precisa usá-lo.
* **Como Funciona**: O proxy intercepta todas as interações até que seja necessário criar a instância real. Depois disso, qualquer operação no proxy é encaminhada para o objeto real.
* **Quando Usar**: Essa estratégia é útil quando você precisa de mais flexibilidade na criação do objeto real ou quando outra parte (como um sistema externo) pode estar envolvida na criação do objeto.

**Por que é uma estratégia?**: Porque é uma abordagem que você escolhe quando precisa de flexibilidade extra. Por exemplo, se o objeto real depende de informações externas que não estão disponíveis no momento da criação do proxy, essa estratégia ajuda a lidar com essa complexidade.

## Ciclo de Vida de Objetos Lentos

O conceito de **ciclo de vida de objetos lentos** refere-se às diferentes fases ou estados pelos quais um objeto lento passa desde o momento em que é criado até que seja completamente inicializado.

**Fases do Ciclo de Vida de Objetos Lentos**

1. **Criação do Objeto Lento**
   * Um objeto pode se tornar "lento" durante sua criação inicial ou mesmo depois de já ter sido criado. Em PHP, você pode usar:
     + ReflectionClass::newLazyGhost() ou ReflectionClass::newLazyProxy() para criar um objeto lento desde o início.
     + ReflectionClass::resetAsLazyGhost() ou ReflectionClass::resetAsLazyProxy() para transformar um objeto existente em um objeto lento.
2. **Estado de Objeto Lento**
   * Após ser criado ou transformado em um objeto lento, ele permanece nesse estado até que sua inicialização seja acionada. Durante esse tempo, o objeto ainda não está totalmente configurado, e suas propriedades podem não ter valores definitivos.
   * Um objeto lento é projetado para parecer um objeto normal, mas na verdade ele adia a execução de sua lógica de inicialização até o momento certo.

**Como a Inicialização é Acionada?**

Um objeto lento pode ser inicializado (ou seja, sua lógica completa pode ser executada) de várias maneiras:

1. **Interação Automática**
   * Se você interagir com o objeto de qualquer forma que observe ou modifique seu estado, isso aciona a inicialização. Isso inclui:
     + Ler ou escrever uma propriedade.
     + Testar se uma propriedade está definida.
     + Serializar o objeto.
     + Iterar sobre as propriedades do objeto.



1. **Marcar Propriedades Como Não-Lentas**

* Você pode tornar um objeto não-lento marcando todas as suas propriedades como "não-lentas". Isso é feito usando:
  + ReflectionProperty::skipLazyInitialization()
  + ReflectionProperty::setRawValueWithoutLazyInitialization()
* Isso garante que as propriedades estejam disponíveis sem acionar a inicialização completa do objeto.

1. **Inicialização Explícita**

* Você pode forçar a inicialização de um objeto lento de forma explícita chamando:
  + ReflectionClass::initializeLazyObject()
  + ReflectionClass::markLazyObjectAsInitialized()
* Isso executa imediatamente toda a lógica de inicialização, transformando o objeto em um objeto normal.

**Importante: Inicialização Completa**

* Os objetos lentos só se tornam totalmente funcionais (não-lentos) quando todas as suas propriedades são marcadas como não-lentas ou quando a lógica de inicialização é acionada.
* Se um objeto não tiver propriedades que possam ser marcadas como não-lentas, os métodos acima não poderão convertê-lo em um objeto normal.

**Exemplo 1: Objeto Fantasma (Lazy Ghost)**

**Contexto do Exemplo**

No exemplo de User, temos um objeto User que só é inicializado com dados (nome e idade) quando acessamos alguma de suas propriedades.

**Situação "Caso Necessário"**

Imagine que você está desenvolvendo uma aplicação de rede social, e o objeto User representa um usuário que está visualizando o feed de notícias. Inicializar todos os dados do usuário (como nome, idade, preferências, etc.) para cada usuário que abre o aplicativo pode ser custoso, especialmente se nem todos esses dados forem usados imediatamente.

**Exemplo de Quando Seria Necessário Inicializar**:

1. **Acessar Dados Específicos**: A inicialização do objeto User acontece quando você acessa a propriedade name ou age pela primeira vez. Por exemplo, se o usuário decidir abrir o perfil completo de outro usuário, você precisará desses dados, e a inicialização é acionada nesse momento.
   * **Cenário Real**: Um botão "Ver Perfil Completo" é clicado. Antes disso, talvez você só precise mostrar um nome de exibição curto ou uma foto de perfil que não requer inicializar o objeto User por completo.

Texto

Descrição gerada automaticamente

**Exemplo 2: Proxy Virtual (Lazy Proxy)**

**Contexto do Exemplo**

No exemplo de Product, o proxy adia a criação do objeto Product com detalhes completos (nome e preço) até que seja necessário.

**Situação "Caso Necessário"**

Imagine que você está desenvolvendo um site de e-commerce. A lista de produtos pode mostrar apenas informações básicas, como o nome e uma imagem pequena. Carregar detalhes completos, como o preço ou descrições detalhadas, para todos os produtos da lista pode ser caro, especialmente se houver muitos produtos na página.

**Exemplo de Quando Seria Necessário Inicializar**:

1. **Visualização de Detalhes**: A inicialização do objeto Product ocorre quando você acessa a propriedade name ou price. Por exemplo, se o usuário clicar no produto para ver mais detalhes (descrição, preço, avaliações), o proxy inicializa o objeto completo.
   * **Cenário Real**: A página de detalhes do produto é aberta, e nesse momento, todos os dados do produto são necessários. Até esse ponto, você pode ter mostrado apenas o nome ou uma imagem sem carregar informações adicionais.

Texto

Descrição gerada automaticamente

**Por Que Isso é Útil?**

Essas estratégias são especialmente úteis para **otimizar o desempenho** e **reduzir o uso de recursos** em aplicações grandes ou que lidam com muitos dados. A inicialização de objetos pesados só é feita quando é realmente necessário, evitando desperdício de memória e processamento.