Funções

Em PHP, as funções são blocos de código que realizam tarefas específicas e podem ser reutilizados em diferentes partes do programa. Elas ajudam a organizar o código, torná-lo mais legível, modular e fácil de manter.

Uma função em PHP é composta por um nome único, seguido de parênteses, onde podem ser passados parâmetros (valores de entrada) necessários para realizar a tarefa. Esses parâmetros são opcionais, dependendo das necessidades da função.

Dentro da função, escrevemos o conjunto de instruções que serão executadas quando a função for chamada. Essas instruções podem realizar cálculos, manipular dados, exibir resultados na tela ou até mesmo interagir com bancos de dados e outras partes do sistema.

Uma função não é executada automaticamente assim que é definida. Ela permanece como um bloco de código reutilizável até que seja chamada explicitamente. Ao escrever o código, podemos definir várias funções, mas elas não executarão suas instruções até que sejam invocadas/chamadas no local apropriado dentro do programa.

Quando queremos usar uma função, fazemos uma chamada a ela pelo seu nome, seguido dos parênteses. Se a função tiver parâmetros, passamos os valores correspondentes dentro dos parênteses. A função é então executada, realiza suas tarefas e, se necessário, pode retornar um valor como resultado.

Além disso, as funções em PHP podem ter escopo, ou seja, um contexto em que elas são definidas e onde as variáveis utilizadas dentro da função são acessíveis. Isso significa que as variáveis definidas dentro de uma função não podem ser acessadas fora dela, a menos que sejam retornadas ou usadas de alguma forma para interagir com o código externo.

As funções em PHP nos permitem dividir o código em partes menores e mais gerenciáveis, o que facilita o trabalho em equipe, a depuração e a reutilização de código. Podemos criar funções para realizar cálculos matemáticos, formatar strings, validar dados, realizar operações em bancos de dados e muito mais.

Portanto, as funções são uma parte fundamental da linguagem PHP, ajudando a criar código mais estruturado, modular e reutilizável, permitindo um desenvolvimento mais eficiente e organizado de projetos.

Funções Criadas pelos Usuários

Uma função em PHP é definida usando a palavra-chave "function" seguida pelo nome da função que deve ser único dentro daquele escopo e descritivo, para indicar a tarefa que ela realiza. o

nome da função deve iniciar com uma letra ou com um underline e diferente dos nomes de variáveis, os nomes de funções não diferenciam maiúsculas e minúsculas, ou seja, você pode chamar a função soma() como: soma(), SOMA(), SomA() e todos serão reconhecidos como a mesma função.

Em seguida, abrimos um par de parênteses e dentro deles podem ser definidos a lista de parâmetros ou valores de entrada que serão utilizados dentro da função para executar sua tarefa. Os parâmetros são como variáveis que podem ser usadas dentro da função. Você pode definir quantos parâmetros forem necessários, separando-os por vírgulas.

O corpo da função é onde você escreve o conjunto de instruções que serão executadas quando a função for chamada. As instruções dentro do corpo da função devem estar delimitadas por chaves "{ }" para indicar o início e o fim da função.

E no corpo da função pode conter um retorno, se a sua função precisa retornar um valor como resultado da sua tarefa, você pode usar a palavra-chave "return" seguida do valor que deseja retornar. O retorno é opcional, e uma função pode não ter nenhum retorno, retornar um único valor ou até mesmo retornar múltiplos valores usando arrays ou objetos.

```php
function calcularSoma($num1, $num2)
{
    // linha 2 até 6 representa o corpo da função
    $soma = $num1 + $num2;
    return $soma; //retorno da função
}
```

O nome da função juntamente com os parênteses abrindo e fechando e a lista de parâmetros que ela recebe formam o que chamamos de assinatura da função, que é utilizada para identificar e uma função no código e os parâmetros necessários a serem passados para que esta função seja executada. Neste exemplo, a assinatura da função "calcularSoma" é:

```php
calcularSoma($num1, $num2)
```

A assinatura indica o nome da função ("calcularSoma") e os parâmetros que ela recebe ("$num1" e "$num2"). A assinatura da função é usada para chamar a função em outros lugares do código. Por exemplo:

```php
$resultado = calcularSoma(5,3);
```

No exemplo acima a função calcularSoma está sendo chamada passando os valores 5 e 3 para a função, estes valores sendo passados são normalmente chamados de argumentos. O parâmetro $num1 está recebendo o valor 5 e o parâmetro $num2 está recebendo o valor 3, exatamente na ordem apresentada na assinatura da função. Após o Corpo da função ser executado o comando

return irá devolver para o local de chamada da função o valor da soma realizada, assim a variável resultado irá receber o valor 8, e poderá imprimir este valor posteriormente.

Criando uma função sem Parâmetros e sem Retorno

Em alguns casos, você pode precisar criar uma função que não recebe nenhum parâmetro e não retorna nenhum valor específico. Essas funções são úteis quando você precisa executar um bloco de código isolado que não depende de valores externos ou quando deseja realizar uma ação específica sem a necessidade de retornar um resultado.

Vamos pensar em no seguinte código:

```php
<?php
echo "João";
echo "<br>";
echo "Maria";
echo "<br>";
echo "Pedro";
echo "<br>";
```

Observe que a cada nome que imprimimos precisamos dar um nova impressão para quebrar a linha no HTML. Podemos extrair este código para uma função, a função **br.**

```php
<?php
function br(){
  echo "<br>";
}
echo "João";
br();
echo "Maria";
br();
echo "Pedro";
br();
```

A função br() não recebe nenhum parâmetro e não retorna nenhum valor, seu objetivo é simplesmente imprimir o código HTML
 fornecendo uma quebra de linha aonde for chamado.

Criando uma função que recebe parâmetros e não retorna valor

Em algumas situações, você pode precisar criar uma função que execute uma ação ou realize algum processamento com base nos parâmetros fornecidos, mas não

precise retornar um valor específico como resultado. Essas funções são úteis quando você deseja modificar variáveis, executar ações específicas ou interagir com outros componentes do sistema sem precisar retornar um resultado.

Continuando o mesmo exemplo gostaríamos de imprimir cada nome como um título h3 e formatado da seguinte forma **>>> nome <<<** mas de modo a reutilizar e diminuir a digitação de código. Para isto iremos criar uma função h3.

```php
<?php
function br(){
  echo "<br>";
}
function h3($nome){
  echo "<h3> >>> {$Nome} <<< </h3>";
}
h3("João");
br();
h3("Maria");
br();
h3("Pedro");
br();
```

Criando uma função que recebe parâmetros e retorna valor

Uma função que recebe parâmetros e retorna um valor é uma estrutura fundamental na programação. Ela permite que você processe dados com base nos parâmetros fornecidos e retorne um resultado específico. Essas funções são úteis quando você precisa realizar cálculos, processamentos ou manipulações nos dados e retornar o resultado desse processamento para uso posterior no programa.

```php
function calcularDobro($numero) {
    $dobro = $numero * 2;
    return $dobro;
}

function calcularTriplo($numero) {
    $triplo = $numero * 3;
    return $triplo;
}

$numero = 5;
$dobro = calcularDobro($numero);
$triplo = calcularTriplo($dobro);
```

```
echo "<h3>O triplo do dobro de $numero é: $triplo</h3>";
```

Nesse exemplo, temos duas funções: calcularDobro e calcularTriplo. A função calcularDobro recebe um número como parâmetro e retorna o dobro desse número. A função calcularTriplo recebe um número como parâmetro e retorna o triplo desse número.

Em seguida, definimos a variável $numero com o valor 5. Chamamos a função calcularDobro($numero) e atribuímos o resultado à variável $dobro. Em seguida, chamamos a função calcularTriplo($dobro) e atribuímos o resultado à variável $triplo.

Por fim, utilizamos o comando echo para imprimir o resultado final em um elemento <h3>. A string exibida contém o valor do triplo do dobro do número original.

Esse exemplo ilustra como podemos passar o resultado de uma função como argumento para outra função, permitindo uma sequência de operações consecutivas. Dessa forma, é possível realizar cálculos ou manipulações em etapas e obter o resultado desejado.

O comando return é usado em uma função para indicar que um valor deve ser retornado como resultado da execução da função. Quando o comando return é encontrado, a execução da função é interrompida e o valor especificado após o return é retornado ao ponto onde a função foi chamada.

Aqui estão alguns pontos importantes sobre o comando return:

- O comando return é seguido pelo valor que desejamos retornar. Esse valor pode ser uma constante, uma variável, uma expressão ou até mesmo o resultado de outra função.
- Pode haver o comando return sem especificar um valor de retorno, desta forma a função irá encerrar sua execução e retornar para o ponto em que foi chamada com o returno nulo (null).
- Ao encontrar o comando return, a função encerra imediatamente sua execução e retorna o valor especificado.
- Uma função pode ter vários pontos de retorno, onde diferentes valores são retornados com base em diferentes condições. No entanto, apenas o primeiro comando return encontrado será executado. Os comandos return subsequentes não serão alcançados ou executados.

```
function verificarParImpar($numero) {
    if ($numero % 2 == 0) {
        return "O número é par";
```

```
    } else {
        return "O número é ímpar";
    }
}

$numero = 6;
$resultado = verificarParImpar($numero); // Chama a função para verificar se
o número é par ou ímpar
echo $resultado; // Saída: O número é par
```

Neste exemplo temos dois comandos return mas apenas um deles será executado dependendo do valor passado e da condição apresentada no comando if.

```
<?php

 function teste(){
      return;
      echo "Teste depois do Retorno";
 }

teste();
```

No Exemplo acima o comando echo nunca será executado pois no momento que o comando return foi executado a função para sua execução e retorna para o ponto em que foi chamada.

Declarando os tipos de dados em funções

Nos exemplos anteriores,não informamos qual é o tipo de dado dos parâmetros da função, isto ocorre porque o PHP é uma linguagem fracamente tipada. Isso significa que o PHP associa automaticamente um tipo de dado a uma variável, dependendo do valor atribuído.

A partir do PHP 7, as declarações de tipo foram introduzidas para as funções e objetos. Essas declarações permitem especificar o tipo de dado esperado para os parâmetros de uma função, bem como o tipo de dado que a função irá retornar. Elas fornecem uma maneira de adicionar mais clareza, consistência e segurança ao código, facilitando a detecção de erros de tipo e tornando o código mais legível e confiável.

Existem quatro tipos de declarações de tipo:

1- Declaração de tipo de parâmetro: Permite especificar o tipo de dado esperado para um parâmetro de função. Por exemplo, podemos definir um parâmetro como int para aceitar apenas valores inteiros.

```php
function somar(int $num1, int $num2) {
    echo $num1 + $num2;
}
```

Caso um valor de um tipo diferente de inteiro seja passado para argumento para esta função e este valor não puder ser convertido para um valor inteiro um erro será apresentado.

```php
somar(5, 3);
echo "<br>";
somar("5", 3);
echo "<br>";
somar(5.8, 3.5);
echo "<br>";
somar("a", 3);
```

Os resultados foram:

```
8
8

Deprecated: Implicit conversion from float 5.8 to int loses precision in D:\xampp\htdocs\exemplos-apostila\exemplo4.php on line 2

Deprecated: Implicit conversion from float 3.5 to int loses precision in D:\xampp\htdocs\exemplos-apostila\exemplo4.php on line 2
8

Fatal error: Uncaught TypeError: somar(): Argument #1 ($num1) must be of type int, string given, called in D:\xampp\htdocs\exemplos-apostila\exemplo4.php on line 14 and defined in D:\xampp\htdocs\exemplos-apostila\exemplo4.php:2 Stack trace: #0 D:\xampp\htdocs\exemplos-apostila\exemplo4.php(14): somar('a', 3) #1 {main} thrown in D:\xampp\htdocs\exemplos-apostila\exemplo4.php on line 2
```

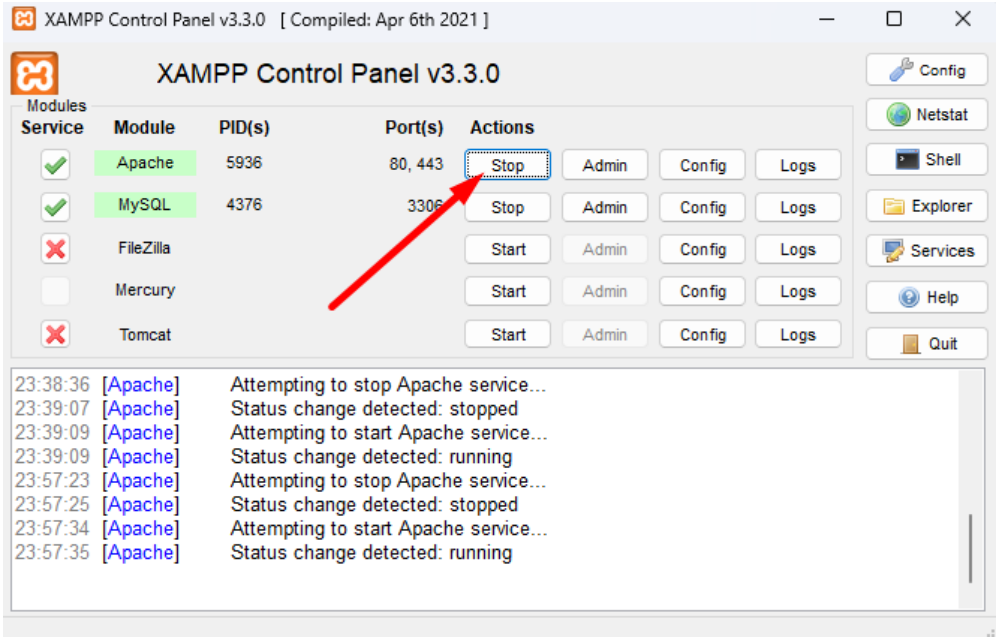
Na primeira chamada dois valores inteiros foram passados e a soma ocorreu corretamente no valor 8. Na segunda chamada, apesar do valor passado para o primeiro parâmetro ter sido uma string ela pôde ser convertida para um número inteiro e a soma ocorreu normalmente resultando no valor 8. Na terceira chamada foram passados os valores como ponto flutuante, ou seja números com casas decimais e um erro foi gerado para cada um dos valores passados, esse erro ocorre no PHP 8 quando há uma conversão implícita de um número em ponto flutuante (float) para um número inteiro (int), resultando em uma perda de precisão. Ele é um aviso de depreciação, o que significa que o recurso em questão está sendo desencorajado e pode ser removido em futuras versões do PHP. Nesse caso, o número em ponto flutuante 5.8 está sendo convertido em um número inteiro. No entanto, como os números inteiros não suportam casas decimais, ocorre uma perda de precisão, resultando em um número inteiro 5. Portanto, o aviso é emitido para informar que a conversão pode não produzir o resultado esperado.
Na chamada 4 uma string “a” foi passada como primeiro parâmetro resultando em um erro Fatal pois o valor passado não pode ser convertido num valor inteiro.

Para resolver o problema encontrado na chamada 3 pode ser utilizada a conversão explícita de tipo como mostrado abaixo:

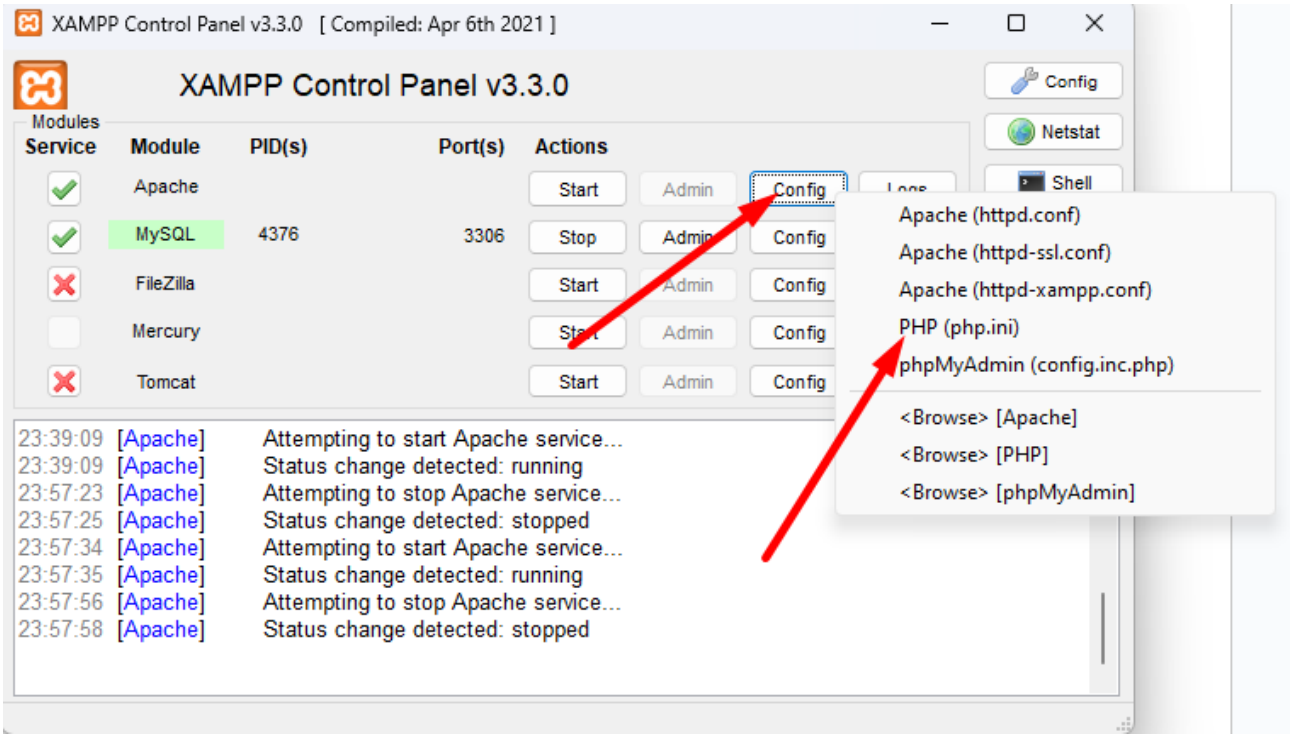
```
somar((int) 5.8, (int) 3.5);
```

Ao colocar (int) antes dos argumentos a serem passados para a função estes serão convertidos para inteiros, assim evitando o erro apresentado.

Este tipo de erro pode não aparecer em todas as instalações do PHP pois depende de como o PHP foi instalado e configurado. Caso não seja exibido em seus testes você pode paralisar o serviço do apache



Depois do Apache parado, clique em config e em seguida em php.ini como mostrado na imagem abaixo:



Procure pela palavra: error_reporting

Esta linha é responsável pela configuração dos tipos de erro que são exibidos no PHP. Caso a sua configuração esteja da forma mostrada abaixo:

error_reporting=E_ALL & ~E_DEPRECATED & ~E_STRICT

modifique para:

error_reporting=E_ALL

Salve o arquivo, feche e inicie novamente o apache. Agora o erro irá aparecer normalmente.

2- Declaração de tipo de retorno: Permite especificar o tipo de dado que a função irá retornar. Por exemplo, podemos declarar que uma função retornará um valor do tipo float. `Para declarar o tipo de retorno de uma função, utiliza-se a sintaxe : tipo após a lista de parâmetros da função e antes do corpo da função. O tipo pode ser um dos tipos primitivos do PHP (como int, float, string, bool) ou até mesmo tipos personalizados (como classes ou interfaces).`

```php
function calcularMedia(float   $nota1, float $nota2, float $nota3): float
{
    $media = ($nota1 + $nota2 + $nota3) / 3;
    return $media;
}
```

```php
$media = calcularMedia(6,5.4,8.2);
echo "A média foi de {$media}";
```

Resultado:

A média foi de 6.5333333333333

Se tentarmos retornar uma string nesta função um erro será produzido

```php
<?php

function calcularMedia(float   $nota1, float $nota2, float $nota3): float
{
    $media = ($nota1 + $nota2 + $nota3) / 3;
    return "A média foi de {$media}";
}

$media = calcularMedia(6,5.4,8.2);
echo $media;
```

Saída:

Fatal error: Uncaught TypeError: calcularMedia(): Return value must be of type float, string returned in D:\xampp\htdocs\exemplos-apostila\exemplo4.php:6 Stack trace: #0 D:\xampp\htdocs\exemplos-apostila\exemplo4.php(9): calcularMedia(6.0, 5.4, 8.2) #1 {main} thrown in **D:\xampp\htdocs\exemplos-apostila\exemplo4.php** on line **6**

3 - Declaração de tipo nullable: Permite que um parâmetro ou retorno de função aceite um tipo de dado específico ou o valor null. Isso é indicado pelo uso do operador ? antes do tipo de dado. Isso é útil quando queremos expressar a possibilidade de um valor estar ausente ou não ter sido definido.

```php
function saudacao(?string $nome) :void {

      if($nome != null){
          echo "<h3>Olá {$nome} !</h3>";
      }else{
        echo "<h3>O nome não foi passado!</h3>";
      }
}

saudacao("Thiago");
saudacao(null);
```

```
saudacao();
```

Observação: O tipo de retorno void significa que esta função não retornar nenhum valor.

Saída:

Olá Thiago !

O nome não foi passado!

Fatal error: Uncaught ArgumentCountError: Too few arguments to function saudacao(), 0 passed in D:\xampp\htdocs\exemplos-apostila\exemplo4.php on line 14 and exactly 1 expected in D:\xampp\htdocs\exemplos-apostila\exemplo4.php:3 Stack trace: #0 D:\xampp\htdocs\exemplos-apostila\exemplo4.php(14): saudacao() #1 {main} thrown in **D:\xampp\htdocs\exemplos-apostila\exemplo4.php** on line **3**

Observe que na terceira chamada da função saudacao nenhum valor foi passado e ocasionou em um erro. Isso ocorre pois apesar da função permitir que valores nulos sejam passados como argumentos, este valor precisa ser especificamente passado. Para resolver este possível problema pode ser utilizado parâmetros opcionais, que veremos em detalhes mais adiante.

```
<?php

function saudacao(?string $nome = null) :void {

      if($nome != null){
          echo "<h3>Olá {$nome} !</h3>";
      }else{
        echo "<h3>O nome não foi passado!</h3>";
      }
}

saudacao("Thiago");
saudacao(null);
saudacao();
```

Neste exemplo na assinatura da função definimos que o parâmetro $nome tem como padrão o valor nulo, ou seja caso não seja passado nenhum valor ele recebe o valor null. Desta forma o erro foi contornado e apresentamos a saída abaixo:

Olá Thiago !

O nome não foi passado!

O nome não foi passado!

4- Declaração de tipo estrita: declaração de tipo estrita, também conhecida como strict typing, é uma funcionalidade introduzida no PHP 7 que permite tornar a verificação de tipos mais rigorosa. Quando a declaração de tipo estrita está ativada, o PHP realiza uma verificação mais precisa dos tipos de dados em tempo de execução, garantindo uma maior consistência e segurança no código.

Para habilitar a declaração de tipo estrita, é necessário adicionar a seguinte instrução no início do arquivo PHP ou no escopo desejado:

```
declare(strict_types=1);
```

Ao habilitar a declaração de tipo estrita, as seguintes regras são aplicadas:

- Verificação estrita de tipos de parâmetros: O PHP verifica se os argumentos passados para uma função correspondem exatamente aos tipos de parâmetros declarados. Se um tipo incompatível for passado, um erro fatal será lançado.
- Verificação estrita de tipos de retorno: O PHP verifica se o valor retornado por uma função corresponde exatamente ao tipo de retorno declarado. Se um tipo incompatível for retornado ou se nenhum valor for retornado em uma função que declara um tipo de retorno, um erro fatal será lançado.
- Conversões de tipo estritas: O PHP não realiza conversões automáticas de tipos incompatíveis. Por exemplo, uma atribuição de uma string para uma variável do tipo int resultará em um erro fatal.

```
<?php
function somar(int $a, int $b) :int {
  return $a + $b;
}
echo somar(5, "5");
?>
```

No exemplo acima a declaração de tipo estrita não está ativada e o segundo parâmetro que é uma string vai tentar ser convertido para número inteiro para que a operação seja realizada. O resultado é apresentado abaixo:

```
10
```

Agora vamos observar o mesmo exemplo com a declaração de tipo estrita ativada:

```php
 <?php

declare(strict_types=1);

function somar(int $a, int $b) :int {
  return $a + $b;
}
echo somar(5, "5");
```

O resultado agora foi um erro fatal como mostrado abaixo:

Fatal error: Uncaught TypeError: somar(): Argument #2 ($b) must be of type int, string given, called in D:\xampp\htdocs\exemplos-apostila\exemplo4.php on line 8 and defined in D:\xampp\htdocs\exemplos-apostila\exemplo4.php:5 Stack trace: #0 D:\xampp\htdocs\exemplos-apostila\exemplo4.php(8): somar(5, '5') #1 {main} thrown in **D:\xampp\htdocs\exemplos-apostila\exemplo4.php** on line **5**

A declaração de tipo estrita ajuda a evitar erros sutis e inesperados relacionados aos tipos de dados. Ela promove a consistência e a clareza no código, facilitando a detecção de problemas em tempo de desenvolvimento. Portanto é recomendado manter a declaração de tipo estrita ativada sempre que possível para promover a robustez e a qualidade do código.

União de Tipos

No PHP, a partir da versão 8, foi introduzido o recurso de união de tipos, que permite especificar múltiplos tipos de dados para parâmetros de função e tipos de retorno. Esse recurso é conhecido como union types.

A união de tipos é extremamente útil quando um parâmetro de função ou o retorno de uma função pode aceitar mais de um tipo de dado. Anteriormente, era necessário usar anotações de tipo genéricas, como mixed ou object, para indicar que um parâmetro ou retorno poderia ser de qualquer tipo. No entanto, a união de tipos oferece uma maneira mais precisa e explícita de definir as possibilidades.

A Sintaxe para utilização da união de tipos no PHP é muito simples. Basta separar os tipos por uma barra |, conforme o exemplo abaixo:

```
function nome(
    tipo1|tipo2|tipoN $nomeparametro1,
    tipo1|tipo2|tipoN $nomeparametro2,
): tipo1|tipo2|tipoN {
    //corpo da função
}
```

Vejamos alguns exemplos para entender melhor como funciona a união de tipos nos parâmetros e retornos de função:

```
function somar(int|float $num1, int|float $num2) : int|float {
    return $num1 + $num2;
}
```

Nesse exemplo, a função somar recebe dois parâmetros $num1 e $num2, que podem ser do tipo int ou float. A união de tipos é indicada pelo uso do caractere de barra vertical (|) entre os tipos. Isso significa que a função pode operar com números inteiros ou de ponto flutuante e retornar um resultado do mesmo tipo.

```
function getValor(int $numero) : int|string|null {
    if ($numero > 0) {
        return $numero * 2;
    } elseif ($numero < 0) {
        return "Negativo";
    } else {
        return null;
    }
}
```

Nesse caso, a função getValor recebe um parâmetro $numero do tipo int e retorna um valor do tipo int, string ou null. Dependendo do valor fornecido, a função pode retornar o dobro do número, a string "Negativo" ou nulo.

A união de tipos nos parâmetros e retornos de função permite uma maior expressividade e precisão na definição dos tipos esperados e retornados. Isso contribui para um código mais legível, além de auxiliar na detecção de erros em tempo de compilação, já que o PHP pode realizar verificações de tipo mais rigorosas.

No entanto, é importante lembrar que a união de tipos deve ser usada com moderação e de forma consciente. É recomendado evitar uniões de tipos excessivamente amplas, pois podem prejudicar a clareza do código e dificultar a manutenção. É preferível usar uniões de tipos apenas quando

necessário e quando faz sentido em relação ao comportamento da função e às necessidades do projeto em questão.

Em resumo, a união de tipos é um recurso poderoso que permite especificar múltiplos tipos de dados para parâmetros e retornos de função no PHP. Com ele, é possível definir com precisão as possibilidades de tipos aceitos e retornados, aumentando a segurança e a clareza do código.

Funções com Parâmetros Opcionais:

As funções com parâmetros opcionais em PHP nos permite definir parâmetros em uma função que podem ser omitidos ao chamar a função. Esses parâmetros têm valores predefinidos, caso nenhum valor seja fornecido durante a chamada da função. Isso proporciona flexibilidade ao desenvolvedor, permitindo que eles personalizem o comportamento da função com base em suas necessidades.

Para criar uma função com parâmetros opcionais, é necessário atribuir um valor padrão ao parâmetro na definição da função. Isso é feito utilizando o operador de atribuição (=) após o tipo de dados e o nome do parâmetro.

É importante ter em mente que os parâmetros opcionais devem ser posicionados sempre por último na lista de parâmetros de uma função. Isso significa que, ao definir uma função com parâmetros opcionais, é necessário garantir que eles sejam os últimos parâmetros listados na assinatura da função.

A razão para isso é que, ao chamar uma função, é possível fornecer argumentos apenas para os parâmetros iniciais, deixando os parâmetros opcionais sem valores específicos. No entanto, se os parâmetros opcionais não forem os últimos da lista, pode ocorrer uma ambiguidade ao tentar determinar qual argumento se destina a qual parâmetro.

```php
<?php
declare(strict_types=1);
function calcularValorTotal(float $preco, int $quantidade = 1, float
$desconto = 0) : float {
    $total = $preco * $quantidade * (1 - $desconto);
    return $total;
}

echo calcularValorTotal(10); // Chama a função com o valor do preço (10) e
usa os valores padrão para os outros parâmetros
echo calcularValorTotal(15, 2); // Chama a função com o valor do preço (15) e
da quantidade (2), usando o valor padrão para o desconto
```

```
echo calcularValorTotal(20, 3, 0.1); // Chama a função com todos os
parâmetros especificados
```

Nesse exemplo, temos a função calcularValorTotal que recebe três parâmetros: $preco, $quantidade e $desconto. O parâmetro $quantidade e o parâmetro $desconto são opcionais e têm valores padrão definidos. Observe que, ao omitir um parâmetro opcional, a função usará o valor padrão definido na declaração da função. No entanto, se os parâmetros opcionais não fossem os últimos na lista de parâmetros da função, ocorreria uma ambiguidade na atribuição dos valores aos parâmetros corretos.

Portanto, é uma prática recomendada seguir a convenção de colocar os parâmetros opcionais por último na lista de parâmetros de uma função para evitar ambiguidades e garantir que a função possa ser chamada corretamente, mesmo omitindo alguns parâmetros opcionais.

Funções com argumentos nomeados

Os argumentos nomeados, também conhecidos como named parameters ou named arguments, são uma funcionalidade introduzida no PHP 8 que permite passar os argumentos para uma função utilizando seus nomes explicitamente. Isso proporciona maior clareza e flexibilidade ao chamar uma função, especialmente quando ela possui um grande número de parâmetros ou quando você deseja especificar apenas alguns parâmetros específicos.

Ao usar argumentos nomeados, você não precisa seguir a ordem dos parâmetros definida na função. Em vez disso, você pode especificar os argumentos utilizando o nome do parâmetro seguido de um dois pontos (:) e o valor correspondente. Isso torna o código mais legível e reduz as chances de erros devido à posição equivocada dos argumentos. Observe a sintaxe abaixo:

```
function nome($parametro1, $parametro2) {}

nome(parametro1: 'valor', parametro2: 'valor');
```

Vamos observar agora o exemplo de cálculo de preço utilizando os argumentos nomeados:

```
<?php
declare(strict_types=1);
function calcularValorTotal(float $preco, int $quantidade = 1, float
$desconto = 0) : float {
    $total = $preco * $quantidade * (1 - $desconto);
    return $total;
}

echo calcularValorTotal(preco: 10, desconto: 0.2);
```

```
echo calcularValorTotal(quantidade:2, desconto: 0.1, preco: 15);
```

Ao chamar essa função, utilizamos a sintaxe de argumentos nomeados para passar os valores dos parâmetros. Isso significa que não precisamos seguir a ordem dos parâmetros definida na função. Em vez disso, especificamos explicitamente o nome do parâmetro seguido por dois pontos (:) e o valor correspondente. No primeiro exemplo de chamada da função estamos passando os valores 10 e 0.2 para os parâmetros preco e desconto, respectivamente. O parâmetro quantidade não é fornecido, portanto, ele usará o valor padrão definido na função, que é 1.

No segundo exemplo de chamada da função estamos fornecendo todos os valores para os parâmetros nomeados, porém sem necessitar seguir a ordem definida na assinatura da função. O valor 2 é atribuído ao parâmetro quantidade, o valor 0.1 é atribuído ao parâmetro desconto e o valor 15 é atribuído ao parâmetro preco.

Saída:

```
8
27
```

O uso de argumentos nomeados torna a chamada da função mais legível e intuitiva, pois podemos identificar claramente qual valor está sendo passado para cada parâmetro, independentemente da ordem. Isso melhora a compreensão do código e ajuda a evitar erros devido à confusão na posição dos argumentos.Os argumentos nomeados podem ser utilizados em qualquer função, tanto as criadas pelo usuário como as disponibilizadas pela linguagem.

Parâmetros Variáveis

No PHP, é possível criar funções que aceitam um número variável de parâmetros usando o recurso de parâmetros variáveis. Esses parâmetros são chamados de parâmetros variáveis ou parâmetros de comprimento variável.

Os parâmetros variáveis permitem que uma função seja definida para aceitar um número arbitrário de argumentos, sem a necessidade de especificar cada argumento individualmente. Isso é especialmente útil quando não sabemos com antecedência quantos argumentos serão passados para a função.

No PHP, os parâmetros variáveis são definidos usando o prefixo ... (três pontos) antes do nome do parâmetro na declaração da função. Isso indica que aquele parâmetro pode aceitar vários valores. Dentro da função, o parâmetro variável é tratado como um array contendo todos os argumentos passados.

Vejamos um exemplo para ilustrar o uso de parâmetros variáveis:

```
function calcularMedia(float ...$notas): float {
```

```
    $soma = 0;
    foreach($notas as $nota){
        $soma += $nota;
    }
    $quantidade = count($notas);

    return $soma / $quantidade;
}

$resultado = calcularMedia(8.5,4.7,6.2);
echo $resultado;
```

Resultado:

```
6.4666666666667
```

Nesse exemplo, temos a função calcularMedia que recebe um número variável de notas como argumentos. Essas notas são representadas pelo parâmetro variável $notas, que é definido com o prefixo ...

Dentro da função, primeiro calculamos a soma de todas as notas passadas como argumentos através de um comando Foreach que passa por todos os elementos do array e vai somando na variável $soma. Em seguida, usamos a função count para obter a quantidade de notas passadas, ou seja, o número de elementos no array $notas.

Finalmente, calculamos a média dividindo a soma das notas pela quantidade de notas. O resultado é retornado como um valor do tipo float.

Na chamada da função calcularMedia(8.5, 4.7, 6.2), estamos passando três notas como argumentos. Essas notas serão agrupadas em um array dentro da função. A função então realiza os cálculos necessários para obter a média das notas fornecidas.

É importante destacar que os parâmetros variáveis podem ser combinados com outros parâmetros normais. No entanto, é recomendado que os parâmetros variáveis sejam sempre os últimos na lista de parâmetros, uma vez que eles capturam todos os argumentos restantes.

Os parâmetros variáveis fornecem flexibilidade na criação de funções que lidam com um número desconhecido de argumentos. Eles são especialmente úteis em casos em que a quantidade de dados a serem processados pode variar, como calcular a média de um conjunto de números ou concatenar uma sequência de strings.

É importante observar que, ao usar parâmetros variáveis, é necessário levar em consideração a segurança e a validação dos dados. Como os argumentos são agrupados em um array, é necessário verificar se os valores fornecidos são do tipo esperado e se estão corretos antes de realizar qualquer operação.

Em resumo, os parâmetros variáveis permitem criar funções no PHP que aceitam um número variável de argumentos. Eles são declarados com o prefixo ... e tratados como um array dentro da

função. Esses parâmetros são úteis quando a quantidade de argumentos pode variar e oferecem maior flexibilidade no design e na implementação de funções.

Escopo de Variáveis

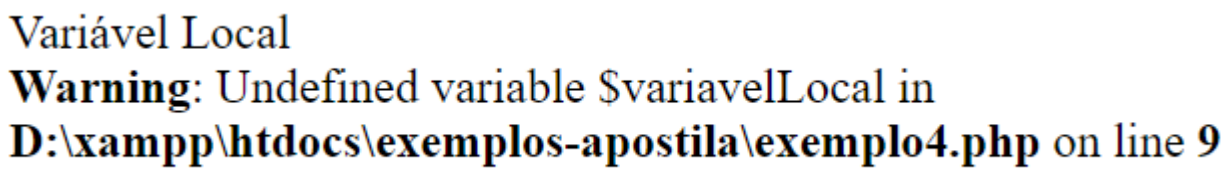
O escopo de variáveis em funções no PHP é um conceito importante a ser compreendido. Ele define a disponibilidade e a visibilidade das variáveis dentro de uma função.

Quando uma variável é declarada dentro de uma função, ela é conhecida como uma variável local. Isso significa que ela só pode ser acessada dentro dessa função específica. As variáveis locais são destruídas e perdem seu valor quando a função termina sua execução. Isso significa que elas não estão disponíveis fora da função.

```php
function exemplo() {
    $variavelLocal = "Variável Local";
    echo $variavelLocal; // Saída: Variável Local
}

exemplo();
echo $variavelLocal; // Erro: variávelLocal não está definida
```

Saída:



Nesse exemplo, a variável $variavelLocal é declarada dentro da função exemplo(). Ela só é acessível dentro dessa função. Se tentarmos acessá-la fora da função, ocorrerá um erro, pois a variável não está definida nesse contexto.

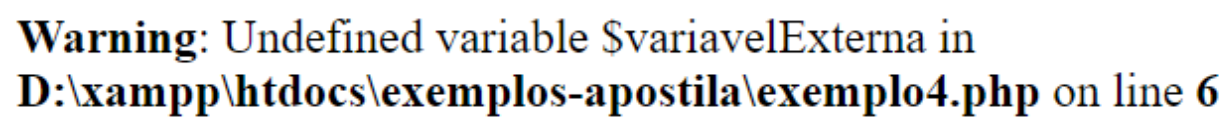
A mesma recíproca é verdadeira, uma variável criada fora de uma função não é visível internamente a ela, a menos, é claro, que seja passada como argumento para a função. Isso ocorre porque as variáveis têm escopo, o que significa que elas só podem ser acessadas em determinadas partes do código onde foram declaradas.

```php
$variavelExterna = "Variável Externa";

function exemplo() {
    echo $variavelExterna; // Erro: variávelExterna não está definida nesta
função
}

exemplo();
```

Saída:



Nesse exemplo, temos uma variável `$variavelExterna` declarada fora da função exemplo(). Dentro dessa função, tentamos acessar a variável diretamente, mas ocorre um erro. Isso acontece porque a função não tem conhecimento da variável externa declarada fora de seu escopo.

Para tornar a variável externa visível dentro da função, podemos passá-la como argumento:

```php
$variavelExterna = "Variável Externa";

function exemplo($variavel) {
    echo $variavel; // Saída: Variável Externa
}

exemplo($variavelExterna);
```

Saída:

Funções com Variáveis Globais

Outra forma de acessar uma variável externa dentro de uma função é através de variáveis globais. que são acessíveis de qualquer lugar do código, incluindo dentro das funções. As variáveis globais são declaradas fora de qualquer função ou escopo específico. O uso da palavra-chave global no PHP permite que você acesse e modifique variáveis globais dentro do escopo de uma função. Aqui está um exemplo:

```php
$variavelGlobal = "Variável Global";

function exemplo() {
    global $variavelGlobal;
    echo $variavelGlobal; // Saída: Variável Global
    $variavelGlobal = "Nova Valor";
}
```

```
exemplo();
echo $variavelGlobal; // Saída: Nova Valor
```

Neste exemplo, temos uma variável global $variavelGlobal definida fora da função exemplo(). Dentro da função, usamos a palavra-chave global seguida pelo nome da variável para indicar que estamos referenciando a variável global.

Dentro da função, podemos acessar e utilizar a variável global normalmente. No exemplo, estamos simplesmente imprimindo o valor da variável global.

Além disso, também podemos modificar o valor da variável global dentro da função. Neste caso, atribuímos um novo valor à variável $variavelGlobal. Após a chamada da função, se imprimirmos o valor da variável global novamente, veremos o novo valor atribuído.

É importante ter cuidado ao usar variáveis globais e a palavra-chave global, pois o uso excessivo de variáveis globais não é recomendado, pois pode tornar o código menos legível e mais difícil de depurar. É uma boa prática limitar o escopo das variáveis ao mínimo necessário e passar os valores necessários como argumentos para as funções, em vez de depender de variáveis globais.

Funções com Variáveis Estáticas

As variáveis estáticas no PHP são variáveis que mantêm seu valor entre as chamadas subsequentes de uma função. Elas são úteis quando queremos rastrear ou manter um estado específico em uma função, mesmo quando ela é chamada várias vezes. Como especialista sênior em PHP, posso explicar com mais detalhes o uso, as vantagens e as melhores práticas para o uso de variáveis estáticas.

Ao declarar uma variável como estática, usamos a palavra-chave static. A variável estática é inicializada apenas uma vez, na primeira chamada da função, e mantém seu valor nas chamadas subsequentes. Isso significa que, ao contrário das variáveis locais, que são destruídas quando a função termina sua execução, as variáveis estáticas preservam seu valor e seu estado entre as chamadas.

```
function contador() {
    static $contagem = 0;
    $contagem++;
    echo $contagem;
}

contador(); // Saída: 1
contador(); // Saída: 2
contador(); // Saída: 3
```

Nesse exemplo, a variável $contagem é declarada como estática usando a palavra-chave static. Ela é inicializada com o valor 0 apenas na primeira chamada da função. Nas chamadas subsequentes, o valor de $contagem é mantido e incrementado a cada chamada. Isso permite que a função mantenha um estado entre as chamadas.

As variáveis estáticas são úteis em várias situações, como:

- Contadores e rastreamento de ocorrências: Elas podem ser usadas para contar quantas vezes uma função foi chamada ou rastrear a ocorrência de determinados eventos em um programa.
- Caching de dados: Podem ser usadas para armazenar valores calculados ou resultados de consultas de banco de dados, evitando a necessidade de recalcular ou consultar os mesmos dados várias vezes.
- Manutenção de estados: Permitem manter um estado específico entre as chamadas de uma função, como manter informações de login, configurações ou estado de uma máquina de estados.

No entanto, é importante lembrar que as variáveis estáticas têm escopo de função, o que significa que elas também não estão disponíveis fora da função.

Passagem de Parâmetros por Referência

Ao trabalhar com funções em PHP, temos duas opções para passar argumentos: por referência ou por valor. Vamos explorar as diferenças entre essas duas abordagens para entender quando usá-las.

- **Passagem por Valor:**
  - Na passagem por valor, é feita uma cópia do valor do argumento para a variável local da função.
  - Qualquer modificação feita na variável dentro da função não afeta a variável original fora da função.
  - Essa é a forma padrão de passagem de argumentos em PHP, caso não seja especificado o uso de referência.
- **Passagem por Referência:**
  - Na passagem por referência, é passada a própria referência do argumento para a função, em vez de uma cópia do valor.
  - Qualquer modificação feita na variável dentro da função afeta diretamente a variável original fora da função.
  - Utiliza-se o operador & para indicar que um parâmetro é passado por referência.

Vejamos um exemplo para ilustrar a diferença entre passagem por valor e por referência:

```
function incrementarValor($valor) {
```

```
    $valor++;
}

function incrementarReferencia(&$valor) {
    $valor++;
}

$num1 = 5;
$num2 = 5;

incrementarValor($num1);
incrementarReferencia($num2);

echo $num1; // Saída: 5 (valor original não foi modificado)
echo $num2; // Saída: 6 (valor foi incrementado por referência)
```

No exemplo acima, temos duas funções: incrementarValor e incrementarReferencia. A primeira recebe um parâmetro por valor e a segunda recebe um parâmetro por referência. Ao chamar essas funções com as variáveis $num1 e $num2, respectivamente, notamos que somente a passagem por referência altera o valor original da variável.

Em geral, a passagem por referência é recomendada apenas quando realmente há necessidade de modificar o valor original do argumento. Para a maioria dos casos, a passagem por valor é suficiente e mais segura, pois evita modificações indesejadas e torna o código mais legível.

Vamos explorar um pouco mais a passagem de parâmetros por referência no PHP. Quando passamos um parâmetro por referência, estamos passando a própria referência da variável para a função, em vez de uma cópia do valor. Isso significa que qualquer alteração feita na variável dentro da função também será refletida no escopo externo.

Vejamos um exemplo para ilustrar a passagem de parâmetros por referência:

```
function dobrarValor(&$numero) {
    $numero *= 2;
}

$valor = 5;
dobrarValor($valor);
echo $valor; // Saída: 10
```

Neste exemplo, definimos a função dobrarValor que recebe um parâmetro por referência usando o operador &. Dentro da função, multiplicamos o valor do parâmetro por 2. Ao chamar a função dobrarValor com a variável $valor, a modificação feita dentro da função é refletida fora dela, e o valor de $valor é alterado para 10.

É importante observar que a passagem de parâmetros por referência só funciona quando a função é chamada e quando a declaração da função especifica que o parâmetro é passado por referência usando o operador &. Portanto, é necessário declarar os parâmetros corretamente tanto na chamada da função quanto em sua definição.

Embora a passagem de parâmetros por referência possa ser útil em certos cenários, como quando precisamos modificar variáveis em escopos externos, é recomendável utilizá-la com cautela. O uso excessivo de parâmetros por referência pode tornar o código mais complexo e difícil de entender e depurar.

Funções Recursivas

As funções recursivas são um recurso poderoso da programação que nos permite resolver problemas dividindo-os em casos menores e tratando cada caso de forma recursiva até atingir uma condição de parada.

Uma função recursiva é uma função que se chama a si mesma dentro de sua própria definição. Isso permite que a função seja executada repetidamente até que uma condição de parada seja alcançada. A recursão é especialmente útil para resolver problemas que podem ser divididos em subproblemas do mesmo tipo.

Vamos ver um exemplo simples de uma função recursiva que calcula o fatorial de um número:

```php
function fatorial($n) {
    // Condição de parada
    if ($n == 0 || $n == 1) {
        return 1;
    } else {
        // Chamada recursiva
        return $n * fatorial($n - 1);
    }
}

echo fatorial(5); // Saída: 120
```

Neste exemplo, a função fatorial recebe um número $n como parâmetro. Se o número for igual a 0 ou 1, a função retorna 1, pois o fatorial desses números é sempre 1. Caso contrário, a função faz uma chamada recursiva, passando $n - 1 como argumento, e multiplica o número atual ($n) pelo fatorial do número anterior. A chamada recursiva continua até que a condição de parada seja alcançada.

As vantagens das funções recursivas em relação às estruturas de repetição, como loops, incluem:

- Simplicidade e legibilidade do código: As funções recursivas permitem expressar soluções de forma mais concisa e intuitiva, especialmente para problemas que possuem uma natureza recursiva intrínseca.

- Divisão de problemas complexos: A recursão permite que um problema complexo seja dividido em subproblemas menores e tratados de forma independente, facilitando a resolução e a compreensão do código.
- Reutilização de código: Funções recursivas podem ser chamadas várias vezes com diferentes argumentos, permitindo a reutilização do código em diferentes contextos.
- Soluções elegantes: Em alguns casos, as soluções recursivas podem ser mais elegantes e eficientes do que as soluções com estruturas de repetição.

No entanto, é importante ter em mente que o uso de funções recursivas requer atenção especial para garantir que a recursão seja finalizada corretamente. Caso contrário, podemos enfrentar problemas de desempenho, como pilha de chamadas excessiva (estouro de pilha), que pode levar a um erro fatal.

Portanto, ao utilizar funções recursivas, é essencial definir corretamente as condições de parada e garantir que o caso base seja atingido em algum momento. Além disso, é fundamental considerar a eficiência da recursão em relação a outras abordagens, especialmente para problemas grandes e complexos.

Em resumo, as funções recursivas são um recurso poderoso da programação que nos permitem resolver problemas de forma elegante e eficiente. Com a devida atenção às condições de parada e ao desempenho, a recursão pode ser uma ferramenta valiosa para solucionar problemas complexos de maneira mais clara e concisa.

Funções Anônimas e Closures

As funções anônimas, também conhecidas como closures, são funções sem nome que podem ser atribuídas a variáveis e chamadas posteriormente. Elas são ideais para situações em que precisamos de funções temporárias ou quando queremos passar funções como argumentos para outras funções.

Aqui está um exemplo de uma função anônima simples:

```php
$mensagem = "Olá, Mundo!";

$funcaoAnonima = function() use (string $mensagem) {
    echo $mensagem;
};

$funcaoAnonima(); // Saída: Olá, Mundo!
```

Neste exemplo, criamos uma função anônima e a atribuímos à variável $funcaoAnonima. A cláusula use permite que a função anônima acesse variáveis externas. No nosso caso, ela acessa a variável $mensagem e a imprime na tela.

As funções anônimas também podem ser passadas como argumentos para outras funções, permitindo a implementação de conceitos como callbacks e programação orientada a eventos. Veja um exemplo:

```php
function executarOperacao($numero1, $numero2, $operacao) {
    $resultado = $operacao($numero1, $numero2);
    echo "O resultado é: " . $resultado;
}

$soma = function($a, $b) {
    return $a + $b;
};

executarOperacao(5, 3, $soma); // Saída: O resultado é: 8
```

Neste exemplo, temos a função executarOperacao, que recebe dois números e uma função anônima de operação. Dentro da função, chamamos a função anônima passando os números e imprimimos o resultado.

Arrow Functions

A partir do PHP 7.4, foi introduzido o conceito de arrow functions (funções de seta). Elas são uma forma mais concisa de escrever funções anônimas simples, especialmente aquelas que consistem em apenas uma expressão.

Aqui está um exemplo de uma arrow function:

```php
$soma = fn($a, $b) => $a + $b;

echo $soma(2, 3); // Saída: 5
```

Neste exemplo, a função de seta $soma recebe dois argumentos e retorna a soma deles diretamente. Não é necessário usar a palavra-chave function ou chaves {} para definir o corpo da função. Isso torna as arrow functions uma opção mais concisa para funções simples e de uma única expressão.

Ao trabalhar com funções anônimas e arrow functions, é importante compreender como elas interagem com o escopo global.

As funções anônimas podem acessar variáveis externas usando a cláusula use, como mostrado no primeiro exemplo. Isso permite que elas acessem e utilizem valores definidos fora do seu escopo.

Já as arrow functions têm um comportamento especial em relação ao escopo. Elas herdam o escopo das variáveis externas automaticamente, sem a necessidade de usar a cláusula use. Isso significa que elas podem acessar variáveis globais sem especificar explicitamente a cláusula use.

As funções anônimas, closures e arrow functions são recursos poderosos do PHP, oferecendo flexibilidade e expressividade ao escrever código. No entanto, é fundamental usá-los com moderação e considerar a clareza e a legibilidade do código. Certifique-se de documentar adequadamente suas funções anônimas e arrow functions para facilitar a compreensão e a manutenção futura do código.