# Visão Geral

Enum são TIPOS que podem ser criados de forma personalizada na hora de programar.

Sabe os “tipos” em programação ? Tipo primitivo int, float... Tipo boolean true or false…

Então, os enums são tipos, mas tipos que você cria.

Quando uma linguagem está sendo criada, existe a possibilidade de os criadores adicionarem recursos que permitem certas coisas naquela linguagem, e uma delas e a enum, ou seja, aumentar o número de tipos que aquela linguagem tem, e esse recurso é oferecido pelos desenvolvedores pela Enum.

O nome **"Enumeration"** vem do conceito de **enumerar** ou **listar** algo. Em programação, uma **enumeration** (ou enum) é um tipo de dado especial que define um conjunto fixo de valores possíveis, como uma lista de opções.

* **Exemplo de Uma Enumeração na Vida Real**: Pense em um semáforo. Ele só tem três estados possíveis: Vermelho, Amarelo, e Verde. Uma enum pode ser usada para representar esses estados de forma clara e limitada.

Cada linguagem, trabalha com enum de forma diferente, no PHP as Enum são como um classe, que armazena os valores da enum dentro dela, e cada chamada da enum, é como se fosse uma “instância” daquela “classe”.

Texto

Descrição gerada automaticamente(ou seja, podemos criar a funcionalidade de enum no PHP sem nem mesmo usar enum, por isso nem é tanto utilizado).

**Exemplo:**

Texto

Descrição gerada automaticamente**==**

**OBS:** Usamos o operador de resolução de escopo “::” para acessar casos em uma enum.

Como mencionei, enums no PHP são internamente representadas como classes, e cada caso de uma enum é como uma constante ou propriedade estática dessa "classe enum".

Embora a palavra "case" apareça tanto em **switch** quanto em **enums**, eles têm **propósitos diferentes.**

**Switch** é mais adequado para lógica de controle, onde você precisa executar diferentes ações com base em valores variáveis. Ele é útil em situações onde o comportamento do sistema muda de acordo com o valor de uma variável.

**Enums** são ideais para definir tipos de dados com valores fixos e limitar as opções disponíveis. Eles são úteis em situações onde você quer garantir que uma variável só pode ter valores específicos, aumentando a segurança e a clareza do código.

Normalmente, podemos ter algumas coisas que podem ser criadas escolhendo um ou outro, que irão resultar no mesmo objetivo definido, porém, podemos diferenciar qual melhor usar e qual seria uma “gambiarra” da seguinte forma:

**Escolhendo Entre switch e Enums**

Embora ambos possam, em certos contextos, chegar a um objetivo semelhante (por exemplo, controlar o comportamento com base em um valor específico), **switch** e **enums** têm propósitos diferentes e, muitas vezes, um é mais apropriado do que o outro. Usar o tipo certo para a situação ajuda a garantir que o código seja **legível, seguro e fácil de manter**.

**Propósitos e Casos de Uso**

1. **switch (Controle de Fluxo)**:
   * **Propósito**: Ideal para controle de fluxo, onde você precisa executar diferentes blocos de código com base em valores variáveis.
   * **Caso de Uso**: Controle dinâmico de comportamento, especialmente quando os valores podem ser variáveis e não são fixos. É ideal para **tomada de decisões**.
2. **Enums (Definição de Tipos)**:
   * **Propósito**: Ideal para definir um conjunto fixo de valores possíveis para uma variável, garantindo que só esses valores sejam aceitos.
   * **Caso de Uso**: Ideal para definir **tipos personalizados com valores restritos**, como estados de um pedido ou tipos de usuário.

**Exemplo: Definindo Permissões de Usuários**

Imagine que você está criando um sistema de permissões para usuários, onde existem diferentes tipos de usuários: **Administrador, Editor, e Visitante**.

1. **Usando Enums (Solução Bem Estruturada)**:
2. Texto

   Descrição gerada automaticamenteAqui, os tipos de usuários são valores fixos que não mudarão com o tempo. Enums são ideais para definir esses valores de forma clara e segura.
   * **Vantagem**: Usar enums para definir TipoUsuario garante que só os valores permitidos (Administrador, Editor, Visitante) possam ser usados. Isso torna o código mais claro e seguro.
3. **Usando Strings Diretas (Gambiarra)**:

* Usar strings para representar os tipos de usuário pode funcionar, mas é mais propenso a erros, pois não há garantia de que o valor seja um dos tipos válidos.

Texto

Descrição gerada automaticamente

**Desvantagem**: Usar strings para tipos de usuário é frágil, pois é fácil cometer erros de digitação e, sem enums, o código não garante que apenas valores válidos sejam usados. Isso pode tornar o código menos seguro e mais propenso a bugs.

**Como Diferenciar Solução Ideal de Gambiarra**

1. **Propósito e Contexto**:
   * Pergunte-se qual é o propósito do código. Se você quer **limitar valores** para garantir segurança e consistência, use enums.
   * Se você quer **executar diferentes ações** com base em valores variados e dinâmicos, use switch.
2. **Segurança e Consistência**:
   * **Solução Ideal**: Enums garantem segurança e consistência ao limitar os valores possíveis.
   * **Gambiarra**: Usar strings ou valores "soltos" em vez de enums reduz a segurança, pois valores inválidos podem facilmente ser usados.
3. **Facilidade de Manutenção**:
   * **Solução Ideal**: Com enums, a manutenção é mais simples, pois os valores permitidos são centralizados e garantidos. Qualquer alteração ou adição é feita diretamente no enum.
   * **Gambiarra**: Usar strings ou constantes soltas pode dificultar a manutenção, já que é fácil esquecer de atualizar todos os lugares que usam esses valores.
4. **Clareza e Legibilidade**:
   * **Solução Ideal**: Enums deixam o código mais claro, pois os valores permitidos são definidos de maneira explícita.
   * **Gambiarra**: Usar valores "soltos" torna o código confuso e pode exigir validações extras para garantir que os valores estão corretos.

**Conclusão**

* **Use enums para definir conjuntos fixos de valores**: Isso é ideal para situações onde você quer garantir que uma variável só pode ter valores específicos, como tipos de usuário, estados de pedido, etc.
* **Use switch para lógica de controle baseada em valores dinâmicos**: Ideal para controlar o fluxo de execução, onde cada caso representa um comportamento específico.

# enum como parâmetro pra funções

Quando temos uma função e precisamos passar o valor de uma enum como parâmetro, podemos:

* passar atribuindo de antemão o ‘case’ da enum em uma variável e depois colocar essa variável dentro do parâmetro...
* Ou podemos simplesmente colocar o acesso ao ‘case’ diretamente dentro do parâmetro.

Das 2 formas funciona perfeitamente.

Como no exemplo:

Texto

Descrição gerada automaticamente

# Backed enums

Mas se por acaso, atribuímos valores a esses cases, da seguinte forma:

Sempre que criamos uma enum, podemos criar os cases, e não atribuir valor nenhum a eles, da seguinte forma:

Texto

Descrição gerada automaticamenteTexto

Descrição gerada automaticamente

Quando temos uma enum sem valores, ela é chamada de Enum Pura, e quanto temos uma Enum com valores, é chamada de Backed Enum (lastreado enum / enum lastreado).

A pergunta que podemos fazer é :

– Ué, mas se a enum funciona como um ‘tipo personalizado’... por que então eu iria informar o tipo de retorno, visto que o tipo de retorno, já é o próprio ‘case’ –

**Enums Puras e Backed Enums:   
Ambos São Tipos Personalizados**

Texto

Descrição gerada automaticamente

1. **Enums Puras**:

Quando você cria uma **enum pura**, você está definindo um **tipo personalizado** que pode ter apenas um conjunto limitado de valores **sem nenhum valor associado**.

**Explicação**: Cor é um **tipo personalizado**, e uma variável desse tipo só pode ter um dos valores Vermelho, Azul, ou Verde.

**2. Backed Enums   
 (Enums com Valores Associados)**:

Quando você cria uma **Backed Enum**, ela **continua sendo um tipo personalizado**, mas cada caso também tem um **valor associado**. Esse valor pode ser uma string ou um int.

Texto

Descrição gerada automaticamente

**Explicação**: NivelAcesso ainda é um **tipo personalizado**, mas agora cada caso (Admin, Editor, Leitor) tem um valor específico associado (1, 2, 3).

***A Diferença Principal***

**OBS:**

* SOMENTE os tipos INT & STRING podem ser usados para indicar o tipo de retorno do valor do tipo.
* Não é aceito valores repetidos.

**Enums Puras**: São apenas **tipos personalizados** sem valores extras. Elas existem para definir opções fixas e claras, sem necessidade de dados associados.

**Backed Enums**: São **tipos personalizados** que, além de definir opções fixas, também associam valores específicos (como strings ou números inteiros) a essas opções.

**Objetos Singleton**

Um **singleton** é um conceito em programação onde uma **classe ou instância** só pode ter **uma única instância** em todo o programa. Isso significa que, se você criar várias variáveis apontando para o mesmo objeto singleton, todas essas variáveis referem-se exatamente à mesma instância.

* No contexto de enums, **cada caso puro** é um **objeto singleton**.
* Texto

  Descrição gerada automaticamenteIsso significa que, quando você usa um caso puro, como Baralho::Copas, ele é sempre a **mesma instância** desse caso, independentemente de quantas vezes você o use ou onde você o referencie.

# Propriedades de Enums no PHP

Dentro das Enums, quando as criamos, temos 3 propriedades padrões que estão “invisíveis” dentro delas, essas são:

Texto

Descrição gerada automaticamente**name**: Retorna o nome do caso como uma string. Útil quando você precisa do nome textual do caso para exibições ou mensagens.

**cases()**: Método estático que retorna todos os casos da enum como um array. Ideal para obter todos os casos da enum e iterar sobre eles.

Texto

Descrição gerada automaticamente

Texto

Descrição gerada automaticamente**value**: (Apenas em Backed Enums) Retorna o valor associado ao caso, como uma string ou número. Usado apenas em enums com valores associados (Backed Enums) quando você precisa do valor subjacente.

# Métodos from() e tryFrom()

**Conceito Geral dos Métodos from() e tryFrom()**

Imagine que você tem um método chamado **from()** que serve para converter ou transformar um valor em algo específico. Da mesma forma, o método **tryFrom()** faz a mesma coisa, mas de uma maneira mais segura, que não causa erros se a conversão falhar.

* **from()**:
  + O objetivo é pegar um valor (como um número ou uma string) e convertê-lo em algo específico. Se a conversão falhar, esse método normalmente lança um erro ou exceção.
* **tryFrom()**:
  + Esse método também tenta fazer uma conversão, mas em vez de lançar um erro se a conversão falhar, ele simplesmente retorna um valor neutro, como null.

**Analogia Simples**

Imagine que você está usando uma **máquina de vendas** que aceita moedas ou notas para liberar um produto. Dependendo de como você usa a máquina, ela se comporta de forma diferente:

1. **from() - Máquina Rigorosa**
   * A máquina espera uma moeda ou nota de um valor exato para liberar o produto. Se você colocar uma moeda ou nota errada, a máquina faz um alarme soar e recusa a transação imediatamente.
   * **Explicação**: O método from() é como essa máquina rigorosa. Ele só aceita valores corretos (casos válidos da enum). Se você tentar usar um valor inválido, ele "faz barulho" e lança um erro.

**2. tryFrom() - Máquina Flexível**

* Essa máquina é mais educada. Se você colocar uma moeda ou nota errada, ela não faz alarme, nem se irrita. Em vez disso, ela simplesmente devolve sua moeda ou nota, sem liberar o produto e sem criar nenhum problema.
* **Explicação**: O método tryFrom() é como essa máquina flexível. Se você fornecer um valor inválido, ele simplesmente retorna null (como se dissesse: "Desculpe, não posso usar isso"), mas não causa um erro.

Ou seja, eles servem basicamente apenas para fazer uma verificação pra lançar um erro, só que um, o erro para o seu código e outro não, simples assim.

Poderiamos fazer isso apenas com if, porém, é melhor usarmos o from() justamente por:

* **Simplicidade e Clareza**
* Usar from() deixa o código mais **limpo e direto**, porque você não precisa escrever verificações manuais complicadas. Ele encapsula a lógica de validação de forma elegante e expressiva.
* Com from(), você está dizendo claramente: "Quero converter este valor em um caso da enum, e se isso não for possível, lance um erro".
* **Validação Automática**
* from() já está embutido na Backed Enum, o que significa que a lógica para verificar se o valor é válido está **previamente implementada pelo PHP**. Isso reduz a chance de erros de implementação na sua própria lógica de verificação.
* **Menos Código, Menos Erros**
* Escrever verificações manuais com if e try-catch pode funcionar, mas aumenta o risco de erros ou omissões. Usando from(), você deixa o PHP cuidar da lógica de validação.

**Usando from() com Enums**

**Como Funciona**: O método from() pega um valor escalar (um int ou string) e tenta encontrar o caso correspondente na enum. Se o valor for válido, ele retorna o caso da enum. Se o valor não for válido, ele lança um erro chamado **ValueError**.

**Quando Usar**: Use from() quando você tem certeza de que o valor é válido e quer garantir que o código pare se o valor não for encontrado.

Texto

Descrição gerada automaticamente

**Usando tryFrom() com Enums**

**Como Funciona**: O método tryFrom() também tenta converter um valor escalar em um caso da enum. A diferença é que, se o valor não for encontrado, ele retorna **null** em vez de lançar um erro.

Texto

Descrição gerada automaticamente**Quando Usar**: Use tryFrom() quando você está lidando com dados que podem ser inválidos e quer evitar que o código pare de funcionar.

# Métodos de enumerações

**$this em Enums ?**

Dentro de um método de uma enum, a variável **$this** se refere à **instância do caso atual** da enum. Isso é similar a como $this se refere ao objeto atual dentro de uma classe.

* Em outras palavras, $this representa o caso específico da enum que está sendo usado no momento.

**Exemplo com $this**

Vamos imaginar que temos uma enum de naipes de cartas:

Texto

Descrição gerada automaticamente

**match com enums ?**

O operador **match** no PHP é uma forma poderosa de executar diferentes blocos de código com base no valor de uma expressão, de maneira semelhante ao switch, mas com algumas vantagens:

* match é mais seguro e claro do que switch.
* Ele retorna um valor, o que o torna útil para atribuições diretas.
* match não precisa de break e verifica exaustivamente todos os casos.

Texto

Descrição gerada automaticamente

Texto

Descrição gerada automaticamenteExemplo com Becked Enums:

Quando trabalhamos com **enums** no PHP, usamos tanto o **operador de resolução de escopo ::** quanto o **operador de objeto ->**, mas cada um tem seu propósito específico.

O operador de resolução de escopo :: é usado para acessar **casos** ou **métodos estáticos** dentro da enum.

O operador de objeto -> é usado para acessar **métodos**.

# Constantes de enumerações

Texto

Descrição gerada automaticamenteEnum não podem ter propriedades variáveis ou estáticas como em classes, mas pode ter constantes, e essas podem ser publicas, protegidas ou privadas.

# Enum com Traits

Texto

Descrição gerada automaticamente**Enums também podem usar traits**, o que permite adicionar funcionalidades reutilizáveis às enums. Traits são blocos de código que você pode incluir em classes ou enums para evitar duplicação de código.

1. **Uso de Traits em Enums**: use Descricao; permite que a enum Naipe use o método descricao() definido na trait Descricao.
2. **$this em Traits**: Dentro da trait, $this se refere ao caso específico da enum que está chamando o método.

**Regras para Enums com Traits**

1. **Enums Podem Usar Traits**: Assim como classes, enums podem incluir traits para reutilizar métodos.
2. **Acesso ao $this**: Dentro da trait, $this se refere à instância do caso da enum, permitindo lógica específica do caso.
3. **Sem Propriedades**: Traits podem adicionar métodos, mas lembre-se de que enums não podem ter propriedades, mesmo se a trait definir propriedades.

# ENUMS E INTERFACES

No PHP, **enums podem implementar interfaces**. Isso significa que você pode garantir que todos os casos de uma enum tenham métodos específicos definidos por uma interface. Ao implementar uma interface, você cria um contrato que a enum deve seguir.

Texto

Descrição gerada automaticamente

Imagine que queremos uma interface chamada Colorido que exige um método cor():

**Explicação**

1. **Implementação da Interface**: enum Naipe implements Colorido faz com que a enum Naipe siga o contrato da interface Colorido, implementando o método cor().
2. **Uso Polimórfico**: Como Naipe implementa Colorido, podemos usar Naipe::Copas em qualquer lugar que aceite um Colorido.

**Regras para Enums com Interfaces**

1. **Enums Podem Implementar Interfaces**: Você pode definir uma ou mais interfaces que a enum deve implementar.
2. **Todos os Métodos Devem Ser Implementados**: Se uma enum implementar uma interface, todos os métodos da interface devem ser implementados na enum.
3. **Tipos de Enums**: Tanto enums puras quanto Backed Enums podem implementar interfaces.

**Regras Importantes Sobre Enums, Interfaces e Traits**

* Herança Não É Permitida em Enums:
  1. Enums no PHP não suportam herança. Isso significa que você não pode estender uma enum com outra enum ou classe.
* Reutilização de Código:
  1. Usar traits é a única forma de reutilizar código em enums, já que a herança não é suportada.
* Implementação de Múltiplas Interfaces:
  1. Uma enum pode implementar várias interfaces, assim como uma classe.
* Backed Enums e Interfaces:
  1. Se uma enum é uma Backed Enum (com valores associados), a declaração da interface deve vir após o tipo de valor.

# Valores enums em expressões constantes

No PHP, podemos usar os cases de uma enum para atribuir ele a uma variável.

Mas não podemos fazer coisas complicadas como usar arrays, chamar funções ou métodos, ou acessar propriedades dentro de constantes.

Texto

Descrição gerada automaticamente

# Diferença entre enum vs objeto

Mesmo que enums sejam baseadas em classes e objetos, elas têm algumas limitações e comportamentos diferentes. Vamos entender isso em termos simples:

**1. Sem Estado ou Propriedades**

Você não pode dar às enums "memória" própria, como faria com objetos que têm propriedades (public $nome; ou private $idade;). Enums não podem ter dados próprios que mudam.

* **Por Que?**: Os casos das enums precisam ser fixos e imutáveis, funcionando mais como **valores constantes** do que como objetos com estado.

**2. Construtores e Destrutores Não São Permitidos**

Você não pode definir um método \_\_construct() ou \_\_destruct() dentro de uma enum.

* **Por Que?**: Porque os casos das enums são sempre **singleton** (ou seja, há apenas uma instância de cada caso), então não faz sentido ter um construtor ou destrutor.

**3. Sem Herança**

Enums não podem estender outras classes ou serem estendidas por outras enums.

* **Por Que?**: Enums são projetadas para serem simples e imutáveis. Herança adicionaria complexidade desnecessária.

**4. Sem Propriedades Estáticas ou de Objeto**

Você não pode ter propriedades dentro de uma enum, nem estáticas nem de objeto. Por exemplo, public static $contador; ou private $nome; não são permitidos.

* **Por Que?**: Isso mantém as enums simples e previsíveis.

**5. Não É Possível Clonar Casos de Enum**

Você não pode duplicar um caso de enum usando o método clone. Cada caso é uma instância única (singleton), e clonar não faz sentido.

* Texto

  Descrição gerada automaticamente com confiança média**Exemplo**:

**6. Métodos Mágicos Limitados**

* A maioria dos **métodos mágicos** não é permitida, como \_\_construct(), \_\_destruct(), e \_\_clone.
* **Métodos Mágicos Permitidos**:
  + \_\_call
  + \_\_callStatic
  + \_\_invoke

**7. Erros de Instanciação**

Você **não pode criar uma instância de um caso de enum usando new**, porque os casos são únicos e pré-definidos, nem com reflection.

**O Que Você Pode Fazer com Enums**

Apesar das limitações, você ainda tem acesso a algumas funcionalidades típicas de objetos:

1. **Métodos**: Você pode ter métodos públicos, privados ou protegidos.
2. **Métodos Estáticos**: Métodos que não dependem de uma instância, como public static function listarTodos().
3. **Constantes**: Você pode definir constantes dentro da enum.
4. **Implementar Interfaces**: Enums podem implementar qualquer número de interfaces para seguir contratos de métodos.
5. **Atributos**: Você pode adicionar atributos (anotações) tanto na enum quanto em casos específicos.
6. **Serialização e desserialização**: Você pode serializar e desserializar uma enum normalmente.

**Constantes Mágicas**

* **::class**: Usada para obter o nome completo da enum, incluindo qualquer namespace.
* **\_\_CLASS\_\_ e \_\_FUNCTION\_\_**: Funcionam normalmente dentro de enums.