# Interfaces predefinidas

Assim como temos classes pré definidas para tratar errors, temos também classes pré definidas para outros objetivos. Essas interfaces e classes existe de forma nativa, justamente pra ajudar na hora do desenvolvimento a evitar ter que escrever tudo do 0, funcionando como ferramentas já prontas.

# Interfaces

As interfaces são contratos que definem quais métodos uma classe deve implementar.   
Temos as seguintes interfaces no PHP:

### Iteradores e Coleções

* **Traversable** (interface raiz para todas as coleções iteráveis)
  + **Iterator**
    - **RecursiveIterator**
    - **OuterIterator**
    - **SeekableIterator**
  + **IteratorAggregate**
* **ArrayAccess** (acesso a objetos como arrays)
* **Countable** (contagem de elementos em objetos)
* **RecursiveCachingIterator** (interface para iteradores recursivos que armazenam o estado de elementos iterados)
* **FilterIterator** (permite criar um iterador que filtra elementos com base em uma lógica personalizada)

### Serialização

* **Serializable** (serialização manual de objetos)
* **JsonSerializable** (personalização de serialização para JSON)

### Funções Anônimas e Callables

* **Throwable** (interface para erros e exceções)
* **Stringable** (representação de objetos como strings)

### Enumeradores

* **UnitEnum** (enumeração básica)
* **BackedEnum** (enumeração com valores associados)

### Observadores

* **SplObserver** (implementa o padrão Observer)
* **SplSubject** (implementa o padrão Observer)

### Streams

* **SessionHandlerInterface** (para manipulação personalizada de sessões)
* **SessionIdInterface** (para manipular IDs de sessão personalizados)
* **SessionUpdateTimestampHandlerInterface** (para controle mais detalhado das sessões)
* **StreamWrapper** (interface usada para criar manipuladores de streams personalizados)

### Banco de Dados

* **PDOStatement** (implementa Iterator para iteração de resultados em PDO)
* **PDORow** (representa uma linha de resultado em PDO)

### Outras interfaces SPL

* **OuterIterator** (uma interface adicional para iteradores avançados, já listada em iteradores)
* **SeekableIterator** (permite pular para posições específicas em uma coleção)

No PHP, interfaces não implementam outras interfaces no sentido técnico ou formal, já que a linguagem não permite que uma interface tenha uma implementação direta de métodos(óbvio). Entretanto, algumas interfaces são projetadas para estarem relacionadas hierarquicamente em termos de propósito ou funcionalidade, criando um padrão de uso consistente. Por exemplo, interfaces como Iterator, RecursiveIterator, OuterIterator e SeekableIterator são todas derivadas funcionalmente do conceito mais amplo representado por Traversable, embora Traversable não seja implementada diretamente, mas sim usada como uma referência conceitual para objetos que podem ser iterados. Assim, as interfaces em PHP estão conectadas logicamente, mas essa relação não se traduz em uma hierarquia de implementação explícita no código

A quantidade de classes que uma interface tem varia, pois tem interfaces que são muito mais importantes e usadas do que outras.

Muitas classes também implementam mais de 1 interface, exemplo, digamos que precisamos de uma classe que faça uma iteração básica porém também precise de uma iteração recursiva em arrays aninhados, logo teremos 1 classe que implementam 2 interfaces.

Essa classe é **RecursiveArrayIterator,** nela as interfaces implementadas são:

* Iterator (permite iteração básica).
* RecursiveIterator (permite iteração recursiva de arrays aninhados).

# interface e seus métodos

Para organizar um estudo ou documentação completa e bem estruturada sobre interfaces e classes no PHP, é mais lógico e coerente abordar interfaces e classes de forma separada, devido à flexibilidade do PHP em permitir que uma classe implemente múltiplas interfaces.

As interfaces no PHP representam contratos. Elas definem métodos que qualquer classe implementadora deve fornecer, mas não têm relação direta com uma única classe.

Algumas classes podem implementar múltiplas interfaces (ex.: ArrayObject implementa ArrayAccess, IteratorAggregate, e Serializable).

Organizar classes diretamente abaixo de uma única interface pode ser confuso, já que essas classes podem pertencer a várias interfaces.

Portanto, veremos primeiro somente os métodos dentro das interfaces.

## Iteradores e Coleções

### **Traversable**

A interface **Traversable** é uma das fundações do PHP para manipular coleções de objetos iteráveis. Ela desempenha um papel central na iteração em estruturas como arrays, objetos e iteradores.

A interface **Traversable** é a raiz para todas as classes e objetos que podem ser iterados com o laço foreach.

**Não pode ser implementada diretamente**: Qualquer classe que queira ser iterável deve implementar uma das interfaces derivadas, como **Iterator** ou **IteratorAggregate**.

**Marcação**: É uma interface "de marcação", ou seja, não define métodos próprios. Sua existência indica que o objeto é iterável.

### **iterator**

A interface **Iterator** é uma das interfaces mais importantes no PHP, fornecendo a base para criar objetos que podem ser iterados com o loop foreach. Ao contrário de **Traversable**, que é apenas uma "interface de marcação", **Iterator** define métodos específicos para controle detalhado de uma iteração.

Iterator é uma interface usada para criar classes que podem ser iteradas. Ela define cinco métodos que controlam como o iterador avança, acessa valores e verifica a validade.

**Usos Comuns**:

* Trabalhar com coleções personalizadas.
* Criar estruturas que encapsulam dados iteráveis, como árvores ou filas.
* Estender iteradores existentes (por exemplo, criando filtros ou transformações).

#### *Métodos da Interface Iterator*

Uma classe que implementa **Iterator** deve definir os seguintes métodos:

1. **current(): mixed**
   * Retorna o elemento atual no iterador.
   * Este método é chamado a cada iteração do loop foreach.
2. **key(): scalar**
   * Retorna a chave do elemento atual.
   * Pode ser um número inteiro ou uma string.
3. **next(): void**
   * Move o cursor para o próximo elemento.
   * É chamado automaticamente pelo foreach após cada iteração.
4. **rewind(): void**
   * Reposiciona o cursor para o início.
   * É chamado automaticamente antes do início de um loop foreach.
5. **valid(): bool**
   * Verifica se a posição atual é válida.
   * Deve retornar true se houver um elemento válido no cursor, ou false caso contrário.

### **recursive iterator**

A interface **RecursiveIterator** no PHP é uma extensão da interface **Iterator** e serve para navegar em estruturas de dados hierárquicas ou recursivas, como árvores, diretórios ou arrays multidimensionais. Ela adiciona métodos específicos para lidar com "filhos" de elementos, permitindo iteração recursiva.

RecursiveIterator é uma interface para criar iteradores que lidam com estruturas recursivas. Cada elemento pode ter "filhos" que também podem ser iterados.

Além dos métodos padrão definidos pela interface **Iterator** (current(), key(), next(), rewind(), valid()), a interface **RecursiveIterator** define dois métodos adicionais:

1. **hasChildren(): bool**
   * **Descrição**: Verifica se o elemento atual tem filhos que podem ser iterados.
   * **Retorno**:
     + true: Se o elemento atual contém filhos.
     + false: Caso contrário.
2. **getChildren(): ?RecursiveIterator**
   * **Descrição**: Retorna um iterador para os filhos do elemento atual.
   * **Retorno**:
     + Uma instância de um objeto que implementa **RecursiveIterator**, caso o elemento atual tenha filhos.
     + null ou nenhum iterador válido, caso não tenha filhos.

### **Outer Iterator**

A interface **OuterIterator** no PHP é usada como uma extensão da interface **Iterator**, permitindo criar iteradores que encapsulam ou "decoram" outro iterador. Essa funcionalidade é útil para manipular, transformar ou aplicar lógica adicional a coleções iteráveis enquanto mantém acesso ao iterador original.

OuterIterator é uma interface que representa um iterador externo. Ele fornece um método adicional, **getInnerIterator()**, que retorna o iterador encapsulado.

**Casos de Uso**:

* Criar decoradores que adicionam funcionalidade a iteradores existentes.
* Encapsular iteradores para fornecer funcionalidades extras, como filtragem, cache ou limites.
* Permitir manipulação indireta de iteradores sem modificar sua implementação original.

#### *Métodos da Interface OuterIterator*

Além dos métodos herdados da interface **Iterator** (current(), key(), next(), rewind(), valid()), a interface **OuterIterator** define o seguinte método adicional:

1. **getInnerIterator(): ?Iterator**
   * **Descrição**: Retorna o iterador encapsulado.
   * **Retorno**:
     + Uma instância de uma classe que implementa **Iterator**, ou null se não houver um iterador interno.

### **seekable iterator**

A interface **SeekableIterator** no PHP é uma extensão da interface **Iterator**, permitindo acesso direto a elementos específicos em uma coleção iterável, como arrays ou listas. Ela é particularmente útil para manipulação de dados onde o posicionamento em um índice específico é necessário.

SeekableIterator é uma interface que permite mover diretamente o cursor de iteração para uma posição específica dentro de uma coleção.

**Casos de Uso**:

* Implementação de coleções que precisam de acesso randômico ou posicionamento direto.
* Manipulação de arquivos ou grandes volumes de dados em segmentos específicos.
* Navegação em resultados paginados ou ordenados.

#### *Métodos da Interface SeekableIterator*

Além dos métodos herdados da interface **Iterator** (current(), key(), next(), rewind(), valid()), a interface **SeekableIterator** define um método adicional:

1. **seek(int $position): void**
   * **Descrição**: Move o cursor para uma posição específica dentro da coleção.
   * **Parâmetro**:
     + $position: O índice para o qual o cursor deve ser movido.
   * **Exceções**:
     + Pode lançar uma exceção se a posição fornecida for inválida.

### **Iterator Aggregate**

A interface **IteratorAggregate** no PHP é uma alternativa à interface **Iterator**, permitindo que uma classe forneça seu próprio iterador para ser usado com foreach. Ao contrário de Iterator, ela não define os métodos de iteração diretamente; em vez disso, exige apenas a implementação de um método chamado **getIterator()** que retorna um objeto que implementa a interface **Traversable** (geralmente um Iterator).

IteratorAggregate é usada para classes que delegam a lógica de iteração para outro objeto.

**Casos de Uso**:

* Ideal para coleções ou classes que já encapsulam um iterador ou uma estrutura de dados iterável.
* Permite maior flexibilidade ao combinar diferentes iteradores em uma única classe.

#### *Método da Interface IteratorAggregate*

A interface **IteratorAggregate** define apenas um método obrigatório:

1. **getIterator(): Traversable**
   * **Descrição**: Retorna um iterador (uma instância de um objeto que implementa Traversable).
   * **Retorno**:
     + Normalmente, um objeto que implementa Iterator ou ArrayIterator.

### **Array Access**

A interface **ArrayAccess** no PHP permite que objetos sejam tratados como arrays. Com ela, você pode acessar e manipular propriedades de um objeto usando a notação de array ($obj['key']), oferecendo flexibilidade e integração com coleções personalizadas.

ArrayAccess é uma interface que permite que objetos respondam a operações de acesso e manipulação típicas de arrays, como leitura, escrita e verificação de índices.

**Casos de Uso**:

* Criar coleções customizadas que se comportam como arrays.
* Implementar estruturas dinâmicas que precisam de controle detalhado sobre como elementos são acessados ou manipulados.
* Interceptar operações de array em objetos, permitindo validações ou transformações.

#### *Métodos da Interface ArrayAccess*

Uma classe que implementa **ArrayAccess** deve implementar os seguintes métodos:

1. **offsetExists($offset): bool**
   * **Descrição**: Verifica se um índice existe no objeto.
   * **Parâmetro**:
     + $offset: O índice a ser verificado.
   * **Retorno**:
     + true: Se o índice existir.
     + false: Caso contrário.
2. **offsetGet($offset): mixed**
   * **Descrição**: Retorna o valor associado a um índice.
   * **Parâmetro**:
     + $offset: O índice cujo valor deve ser retornado.
   * **Retorno**:
     + O valor associado ao índice.
3. **offsetSet($offset, $value): void**
   * **Descrição**: Define um valor para um índice específico.
   * **Parâmetros**:
     + $offset: O índice que será definido.
     + $value: O valor a ser associado ao índice.
   * **Nota**:
     + Se $offset for null, a implementação deve adicionar o valor como um novo elemento (como faria um array normal).
4. **offsetUnset($offset): void**
   * **Descrição**: Remove um índice do objeto.
   * **Parâmetro**:
     + $offset: O índice a ser removido.

#### **Countable**

A interface **Countable** no PHP permite que objetos implementem a função **count()**, que retorna o número de elementos de uma coleção ou estrutura. É amplamente utilizada em coleções personalizadas, onde o número de elementos precisa ser determinado de forma dinâmica ou controlada.

A interface **Countable** exige que as classes implementem um único método, **count()**, que deve retornar um número inteiro representando o tamanho da coleção.

**Casos de Uso**:

* Implementar coleções personalizadas que precisam de contagem específica.
* Representar objetos que encapsulam conjuntos de dados, listas ou filas.

#### *Método da Interface Countable*

Uma classe que implementa **Countable** deve definir o seguinte método:

1. **count(): int**
   * **Descrição**: Retorna o número de elementos no objeto.
   * **Retorno**:
     + Um número inteiro representando o número de elementos.

### **Recursive Caching Iterator**

A interface **RecursiveCachingIterator** é uma extensão do **CachingIterator**, combinada com a funcionalidade de **RecursiveIterator**. Essa interface permite iterar sobre estruturas recursivas (como árvores ou arrays multidimensionais) enquanto armazena os estados já iterados, facilitando operações que dependem de histórico ou repetição de elementos.

O **CachingIterator** é uma **classe** no PHP, não uma interface.

A interface RecursiveCachingIterator permite criar iteradores recursivos que armazenam o estado dos elementos durante a iteração. Ela é útil para operações que requerem acesso ao histórico de elementos ou que dependem de verificações repetidas.

**Casos de Uso**:

* Iteração recursiva sobre estruturas hierárquicas.
* Armazenamento e reutilização de estados durante a iteração.
* Implementação de lógica condicional baseada no histórico de iteração.

#### *Métodos da Interface RecursiveCachingIterator*

Como **RecursiveCachingIterator** combina as funcionalidades de **CachingIterator** e **RecursiveIterator**, ela herda os métodos de ambas.

#### *Métodos de CachingIterator*

1. **hasNext(): bool**
   * Retorna true se houver um próximo elemento no iterador.
2. **getFlags(): int**
   * Retorna as flags de comportamento do iterador.
3. **setFlags(int $flags): void**
   * Define as flags de comportamento do iterador.
4. **\_\_toString(): string**
   * Retorna o valor atual como string (se implementado).
5. **getCache(): array**
   * Retorna um array contendo os elementos já iterados.

#### *Métodos de RecursiveIterator*

1. **hasChildren(): bool**
   * Retorna true se o elemento atual tiver filhos que podem ser iterados recursivamente.
2. **getChildren(): ?RecursiveIterator**
   * Retorna um iterador para os filhos do elemento atual, ou null se não houver filhos.

### **Filter Iterator**

A classe **FilterIterator** no PHP é uma classe abstrata da SPL que permite criar iteradores que filtram elementos de outro iterador com base em condições definidas pelo desenvolvedor. Ela funciona como um "decorador", encapsulando outro iterador e fornecendo apenas os elementos que atendem aos critérios especificados.

FilterIterator é uma classe abstrata que estende a classe **IteratorIterator** e permite implementar lógica personalizada para filtrar elementos durante a iteração.

**Casos de Uso**:

* Filtrar elementos de arrays ou coleções com base em regras específicas.
* Criar pipelines de dados onde apenas os elementos desejados são processados.
* Aplicar validações em tempo de iteração.

#### *Métodos da Classe FilterIterator*

A classe **FilterIterator** herda os métodos da classe **IteratorIterator** e adiciona um método abstrato que deve ser implementado em subclasses:

1. **\_\_construct(Iterator $iterator)**
   * Cria uma nova instância de FilterIterator.
   * **Parâmetro**:
     + $iterator: O iterador a ser filtrado.
2. **accept(): bool**
   * **Descrição**: Método abstrato que define a lógica de filtragem.
   * **Retorno**:
     + true: O elemento atual é aceito e será incluído na iteração.
     + false: O elemento atual será ignorado.
   * Deve ser implementado em qualquer classe derivada de FilterIterator.
3. **Métodos Herdados de IteratorIterator**:
   * getInnerIterator(): Retorna o iterador encapsulado.
   * current(), key(), next(), rewind(), valid(): Métodos padrão de um iterador.

## serialização

### **Serializable**

A interface **Serializable** no PHP permite que objetos personalizem a forma como são serializados e desserializados. Isso é útil para controlar quais dados de um objeto devem ser convertidos em uma string (serialização) e como reconstruir o objeto a partir dessa string (desserialização).

A interface **Serializable** permite que uma classe defina métodos para controlar explicitamente o comportamento de serialização e desserialização de seus objetos.

**Casos de Uso**:

* Serializar objetos complexos para armazenamento em cache, bancos de dados ou arquivos.
* Restringir quais dados de um objeto são serializados.
* Manter compatibilidade ao desserializar objetos, mesmo se a classe for alterada.

#### *Métodos da Interface Serializable*

Uma classe que implementa **Serializable** deve implementar os seguintes métodos:

1. **serialize(): string**
   * **Descrição**: Define como os dados do objeto serão convertidos em uma string durante a serialização.
   * **Retorno**:
     + Uma string que representa o estado do objeto.
2. **unserialize(string $data): void**
   * **Descrição**: Reconstrói o objeto a partir de uma string serializada.
   * **Parâmetros**:
     + $data: A string gerada pelo método serialize.

### **Json Serializable**

A interface **JsonSerializable** no PHP permite que objetos definam como devem ser serializados em formato JSON ao serem usados com a função **json\_encode()**. Isso é particularmente útil para personalizar a saída de objetos complexos ou controlar quais dados serão incluídos na serialização.

JsonSerializable é usada para classes que precisam customizar sua representação em JSON.

* **Casos de Uso**:
  + Serializar objetos de forma customizada em APIs REST.
  + Controlar quais dados de um objeto devem ser incluídos em respostas JSON.
  + Manter compatibilidade e simplicidade ao compartilhar dados entre sistemas.

#### *Método da Interface JsonSerializable*

Uma classe que implementa **JsonSerializable** deve implementar o seguinte método:

1. **jsonSerialize(): mixed**
   * **Descrição**: Define como o objeto será convertido em dados JSON.
   * **Retorno**:
     + O valor retornado pode ser qualquer dado válido para JSON: array, objeto, string, número, booleano ou null.

## Funções Anônimas e Callables

### **Throwable**

A interface **Throwable** foi introduzida no PHP 7 para unificar o tratamento de erros e exceções. Ela serve como a interface base para as classes **Exception** e **Error**, permitindo que ambos sejam tratados de maneira uniforme com o bloco try-catch.

Throwable é a interface raiz para todas as exceções (**Exception**) e erros (**Error**) no PHP. Qualquer classe que implemente Throwable pode ser lançada e capturada em um bloco try-catch.

**Casos de Uso**:

* Garantir que todos os objetos lançáveis sigam um contrato comum.
* Unificar o tratamento de exceções e erros fatais.

#### *Métodos da Interface Throwable*

A interface **Throwable** define métodos obrigatórios que devem ser implementados por qualquer classe que a implemente:

1. **getMessage(): string**
   * **Descrição**: Retorna a mensagem associada ao erro ou exceção.
   * **Exemplo**: "Division by zero"
2. **getCode(): int**
   * **Descrição**: Retorna o código associado ao erro ou exceção (geralmente definido pelo desenvolvedor).
3. **getFile(): string**
   * **Descrição**: Retorna o caminho do arquivo onde o erro ou exceção foi lançado.
4. **getLine(): int**
   * **Descrição**: Retorna o número da linha onde o erro ou exceção foi lançado.
5. **getTrace(): array**
   * **Descrição**: Retorna o rastreamento da pilha no momento em que o erro ou exceção foi lançado.
6. **getTraceAsString(): string**
   * **Descrição**: Retorna o rastreamento da pilha como uma string.
7. **\_\_toString(): string**
   * **Descrição**: Retorna uma representação textual completa do erro ou exceção, incluindo mensagem, código, arquivo e rastreamento da pilha.

### **Stringable**

A interface **Stringable**, introduzida no **PHP 8.0**, representa objetos que podem ser convertidos para string. Ela é automaticamente implementada por qualquer classe que define o método mágico **\_\_toString()**, permitindo que tais objetos sejam usados de forma natural em contextos que esperam strings.

A interface **Stringable** fornece um contrato que garante que um objeto pode ser representado como uma string. Qualquer classe que implementa ou herda **\_\_toString()** implementa implicitamente a interface Stringable.

**Casos de Uso**:

* Criar classes que podem ser tratadas como strings em funções nativas ou operações de concatenação.
* Garantir compatibilidade com APIs que esperam strings.

#### *Método Associado*

A interface **Stringable** não define novos métodos diretamente, mas está vinculada ao método mágico **\_\_toString()**. Para que uma classe seja considerada Stringable, ela deve implementar:

1. **\_\_toString(): string**
   * **Descrição**: Retorna a representação textual do objeto.
   * **Retorno**:
     + Uma string que representa o estado ou a lógica definida pelo objeto.

## Enumeradores

### **Unit Enum**

A interface **UnitEnum**, introduzida no **PHP 8.1**, é usada como base para todas as **enumerações** (enums) no PHP. Ela define comportamentos e métodos padrão para classes que utilizam o recurso de enums, permitindo representar valores constantes e garantir uma interface consistente para acessar esses valores.

UnitEnum é a interface que todas as enums do PHP implementam automaticamente. Ela fornece o método **cases()** para obter todas as instâncias de uma enum.

**Casos de Uso**:

* Representar estados fixos ou valores constantes de forma segura e padronizada.
* Substituir conjuntos de constantes (define ou const) com mais funcionalidade e controle.

#### *Método da Interface UnitEnum*

A interface **UnitEnum** define o seguinte método obrigatório:

1. **cases(): array**
   * **Descrição**: Retorna todas as instâncias da enum como um array.
   * **Retorno**:
     + Um array contendo todas as instâncias da enum (os casos).

### **Backed Enum**

A interface **BackedEnum**, introduzida no **PHP 8.1**, estende a funcionalidade de **UnitEnum**, permitindo que enums tenham valores associados. Esses valores podem ser do tipo **string** ou **int**, tornando as enums mais versáteis para representar estados ou constantes com valores significativos.

BackedEnum é usada para enums que precisam ter um valor associado a cada caso. Esse valor é chamado de **valor escalável** e deve ser único dentro da enum.

**Casos de Uso**:

* Representar estados ou tipos que têm um identificador numérico ou textual.
* Garantir compatibilidade com sistemas externos que usam valores fixos para estados (ex.: bancos de dados, APIs).

#### *Métodos da Interface BackedEnum*

Além dos métodos herdados de **UnitEnum**, BackedEnum define os seguintes métodos:

1. **from(string|int $value): static**
   * **Descrição**: Retorna o caso da enum correspondente ao valor fornecido.
   * **Parâmetros**:
     + $value: O valor escalável associado ao caso.
   * **Retorno**:
     + O caso correspondente.
   * **Exceção**:
     + Lança uma ValueError se o valor fornecido não corresponder a nenhum caso.
2. **tryFrom(string|int $value): ?static**
   * **Descrição**: Tenta retornar o caso da enum correspondente ao valor fornecido, sem lançar exceção em caso de falha.
   * **Parâmetros**:
     + $value: O valor escalável associado ao caso.
   * **Retorno**:
     + O caso correspondente ou null se o valor não for válido.

## Observadores

### **Spl Observer**

A interface **SplObserver** faz parte da SPL (**Standard PHP Library**) no PHP e é usada para implementar o padrão de projeto **Observer**. Esse padrão é amplamente utilizado para estabelecer uma relação entre objetos onde um (o **sujeito**) notifica os outros (**observadores**) sobre mudanças de estado ou eventos.

A interface SplObserver define o comportamento que um **observador** deve ter em um padrão **Observer**. Qualquer classe que implemente essa interface pode ser registrada como observadora de um **sujeito** que implementa a interface **SplSubject**.

**Casos de Uso:**

* **Monitorar mudanças de estado em objetos.**
* **Implementar sistemas de eventos ou notificações.**
* **Criar relações desacopladas entre objetos.**

#### *Método da Interface SplObserver*

A interface **SplObserver** exige a implementação de apenas um método:

1. **update(SplSubject $subject): void**
   * **Descrição**: Esse método é chamado pelo **sujeito** quando ocorre uma mudança de estado ou evento.
   * **Parâmetros**:
     + $subject: O sujeito que notificou o observador. Isso permite ao observador acessar informações adicionais sobre a mudança.
   * **Retorno**:
     + Nenhum retorno (void).

### **Spl Subject**

A interface **SplSubject** faz parte da SPL (**Standard PHP Library**) no PHP e é usada para implementar o padrão de projeto **Observer**. No contexto do padrão, o **sujeito** (ou subject) é o objeto que mantém uma lista de observadores e os notifica sobre mudanças de estado ou eventos.

SplSubject define os métodos que um **sujeito** deve implementar no padrão Observer. Ele gerencia a lista de observadores e se encarrega de notificá-los quando necessário.

**Casos de Uso**:

* Implementar sistemas de eventos e notificações.
* Criar objetos que possam ser monitorados por outros componentes desacoplados.
* Notificar múltiplos observadores sobre mudanças de estado ou eventos.

#### *Métodos da Interface SplSubject*

Uma classe que implementa **SplSubject** deve definir os seguintes métodos:

1. **attach(SplObserver $observer): void**
   * **Descrição**: Adiciona um observador à lista de observadores do sujeito.
   * **Parâmetro**:
     + $observer: O objeto que implementa a interface SplObserver.
2. **detach(SplObserver $observer): void**
   * **Descrição**: Remove um observador da lista.
   * **Parâmetro**:
     + $observer: O objeto que implementa a interface SplObserver.
3. **notify(): void**
   * **Descrição**: Notifica todos os observadores registrados sobre uma mudança de estado.
   * **Parâmetros**: Nenhum.
   * **Comportamento**:
     + Chama o método update() de cada observador registrado.

## Streams

### **Session Handler Interface**

A interface **SessionHandlerInterface** no PHP permite criar manipuladores personalizados de sessões. Com ela, você pode substituir o comportamento padrão do PHP para armazenamento de sessões (geralmente arquivos no sistema) por soluções personalizadas, como armazenamento em bancos de dados, caches (Redis, Memcached), ou qualquer outro sistema.

A interface **SessionHandlerInterface** define um conjunto de métodos que o manipulador de sessões deve implementar. Esses métodos permitem ao desenvolvedor gerenciar diretamente a abertura, leitura, gravação e exclusão de dados de sessão.

**Casos de Uso**:

* Personalizar o armazenamento de sessões (ex.: usar bancos de dados ou caches).
* Adicionar criptografia aos dados de sessão.
* Integrar sessões com sistemas distribuídos.

#### Métodos da Interface SessionHandlerInterface

Qualquer classe que implementa **SessionHandlerInterface** deve implementar os seguintes métodos:

1. **open(string $path, string $name): bool**
   * **Descrição**: Inicializa o manipulador de sessões.
   * **Parâmetros**:
     + $path: O caminho onde os dados da sessão devem ser armazenados.
     + $name: O nome da sessão.
   * **Retorno**:
     + true em caso de sucesso, ou false em caso de falha.
2. **close(): bool**
   * **Descrição**: Fecha o manipulador de sessões, geralmente para liberar recursos.
   * **Retorno**:
     + true em caso de sucesso, ou false em caso de falha.
3. **read(string $id): string**
   * **Descrição**: Lê os dados de uma sessão com base no ID fornecido.
   * **Parâmetro**:
     + $id: O ID da sessão.
   * **Retorno**:
     + Os dados da sessão como string, ou uma string vazia se não houver dados.
4. **write(string $id, string $data): bool**
   * **Descrição**: Escreve os dados de uma sessão com base no ID fornecido.
   * **Parâmetros**:
     + $id: O ID da sessão.
     + $data: Os dados a serem armazenados.
   * **Retorno**:
     + true em caso de sucesso, ou false em caso de falha.
5. **destroy(string $id): bool**
   * **Descrição**: Remove os dados de uma sessão.
   * **Parâmetro**:
     + $id: O ID da sessão a ser destruída.
   * **Retorno**:
     + true em caso de sucesso, ou false em caso de falha.
6. **gc(int $max\_lifetime): int|false**
   * **Descrição**: Executa a coleta de lixo para limpar sessões expiradas.
   * **Parâmetro**:
     + $max\_lifetime: O tempo máximo de vida das sessões (em segundos).
   * **Retorno**:
     + O número de sessões removidas ou false em caso de falha.

### **Session Id Interface**

A interface **SessionIdInterface** no PHP foi introduzida no **PHP 7.1** e permite personalizar como os IDs de sessão são gerados, validados e manipulados. Ela é uma extensão do sistema de gerenciamento de sessões e é usada em conjunto com **SessionHandlerInterface**.

SessionIdInterface permite ao desenvolvedor definir um manipulador personalizado para criar e validar IDs de sessão. Isso é útil para implementar métodos específicos de geração de IDs, como usar UUIDs ou integrar sistemas de autenticação distribuídos.

**Casos de Uso**:

* Gerar IDs de sessão mais seguros.
* Garantir compatibilidade com sistemas externos que exigem formatos específicos para IDs.
* Implementar verificações adicionais para IDs de sessão.

#### *Métodos da Interface SessionIdInterface*

Uma classe que implementa **SessionIdInterface** deve definir os seguintes métodos:

1. **create\_sid(): string**
   * **Descrição**: Gera um novo ID de sessão único.
   * **Retorno**:
     + Uma string que representa o novo ID de sessão.

### **Session-Update-Times-tamp-Handler-Interface**

A interface **SessionUpdateTimestampHandlerInterface**, introduzida no **PHP 7.0**, permite maior controle sobre a atualização do timestamp das sessões. Ela é útil para sistemas que precisam de gerenciamento personalizado da expiração de sessões ou que desejam controlar quando uma sessão ativa deve ser atualizada.

A interface **SessionUpdateTimestampHandlerInterface** é projetada para permitir que desenvolvedores controlem como e quando o timestamp de uma sessão é atualizado. Isso é especialmente útil em sistemas que têm requisitos específicos de expiração de sessão.

**Casos de Uso**:

* Implementar um sistema de gerenciamento de sessão que atualiza timestamps de forma customizada.
* Evitar a atualização automática de sessões inativas.
* Controlar manualmente a renovação de sessões ativas em sistemas com alta segurança.

#### *Métodos da Interface Session Update Timestamp Handler Interface*

Qualquer classe que implemente **SessionUpdateTimestampHandlerInterface** deve definir os seguintes métodos:

1. **validateId(string $id): bool**
   * **Descrição**: Valida se o ID da sessão fornecido é válido.
   * **Parâmetro**:
     + $id: O ID da sessão a ser validado.
   * **Retorno**:
     + true se o ID for válido, ou false caso contrário.
2. **updateTimestamp(string $id, string $data): bool**
   * **Descrição**: Atualiza o timestamp de uma sessão com base no ID e nos dados fornecidos.
   * **Parâmetros**:
     + $id: O ID da sessão a ser atualizada.
     + $data: Os dados associados à sessão.
   * **Retorno**:
     + true se a atualização for bem-sucedida, ou false caso contrário.

### **Stream Wrapper**

No PHP, um **Stream Wrapper** é uma interface que permite manipular fluxos de dados como se fossem arquivos, mas estendendo a funcionalidade para incluir protocolos e sistemas personalizados, como acesso a URLs, bancos de dados, ou até mesmo memória. É uma ferramenta poderosa para criar abstrações sobre fontes de dados diversas.

Um Stream Wrapper traduz operações de fluxo padrão (como leitura, escrita, busca, etc.) em funcionalidades específicas para uma fonte de dados ou protocolo.

**Casos de Uso**:

* Manipular recursos remotos (ex.: HTTP, FTP) como se fossem arquivos locais.
* Criar sistemas personalizados de armazenamento (ex.: memória, bancos de dados).
* Implementar protocolos virtuais para acessar dados em formatos específicos.

#### *Principais Métodos de Stream Wrapper*

Ao implementar um Stream Wrapper, você deve definir métodos que correspondem às operações de fluxo padrão. Os principais métodos são:

1. **Abertura e Fechamento**
   * stream\_open(string $path, string $mode, int $options, string &$opened\_path): bool
   * stream\_close(): void
2. **Leitura e Escrita**
   * stream\_read(int $count): string
   * stream\_write(string $data): int
3. **Busca e Tamanho**
   * stream\_seek(int $offset, int $whence): bool
   * stream\_tell(): int
   * stream\_eof(): bool
   * stream\_stat(): array
4. **Manipulação de Diretórios (se aplicável)**
   * dir\_opendir(string $path, int $options): bool
   * dir\_readdir(): string|false
   * dir\_closedir(): bool
5. **Outros Métodos**
   * unlink(string $path): bool
   * rename(string $path\_from, string $path\_to): bool
   * mkdir(string $path, int $mode, int $options): bool
   * rmdir(string $path, int $options): bool

***Tipos de Wrappers de Fluxo***

1. **Wrappers Padrão**
   * Texto

     Descrição gerada automaticamenteO PHP inclui vários wrappers de fluxo padrão, como:

**2. Wrappers Personalizados**

* + Criados pelo desenvolvedor para implementar protocolos ou abstrações customizadas.

## Outras interfaces SPL

### **Outer Iterator**

A interface **OuterIterator** no PHP faz parte da SPL (**Standard PHP Library**) e é projetada para encapsular outro iterador, permitindo implementar lógica adicional ou modificar o comportamento do iterador subjacente. Ela é usada principalmente para criar "decoradores" de iteradores, que adicionam funcionalidades extras sem modificar diretamente o iterador original.

OuterIterator é uma interface que permite criar iteradores que decoram outros iteradores. O iterador encapsulado pode ser acessado diretamente através do método **getInnerIterator()**.

**Casos de Uso**:

* Adicionar funcionalidades extras a iteradores existentes.
* Modificar ou filtrar os elementos de outro iterador.
* Encadear iteradores para criar fluxos de dados complexos.

#### *Métodos da Interface OuterIterator*

Uma classe que implementa **OuterIterator** deve implementar o seguinte método, além dos métodos padrão da interface **Iterator**:

1. **getInnerIterator(): Traversable**
   * **Descrição**: Retorna o iterador interno encapsulado.
   * **Retorno**:
     + O iterador subjacente que está sendo decorado.

### SeekableIterator

A interface **SeekableIterator** no PHP faz parte da SPL (**Standard PHP Library**) e é usada para iteradores que permitem "buscar" diretamente uma posição específica dentro de uma sequência. Ela estende a funcionalidade básica da interface **Iterator** ao adicionar um método para mover o ponteiro para qualquer posição válida na coleção.

SeekableIterator é uma interface para iteradores que oferecem acesso aleatório aos elementos, permitindo navegar diretamente para qualquer índice válido sem precisar percorrer os elementos sequencialmente.

**Casos de Uso**:

* Navegar diretamente para um elemento específico em grandes coleções.
* Criar iteradores que simulam comportamentos de listas ou arrays.
* Implementar controle de navegação eficiente em coleções baseadas em índices.

#### *Métodos da Interface SeekableIterator*

Além dos métodos padrão da interface **Iterator**, como current(), next(), key(), rewind(), e valid(), o **SeekableIterator** adiciona:

1. **seek(int $position): void**
   * **Descrição**: Move o ponteiro do iterador para a posição especificada.
   * **Parâmetro**:
     + $position: A posição para a qual o ponteiro deve ser movido.
   * **Comportamento**:
     + Lança uma exceção (OutOfBoundsException) se a posição não for válida.