

# Programa de Formação

# Web Developers PHP



**CONSTRUSITE BRASIL**  
O SEU SUCESSO ONLINE

**GALAXPAY**   
SEU NEGÓCIO RECORRENTE



01

...

## P00

Programação Orientada a  
Objetos

02

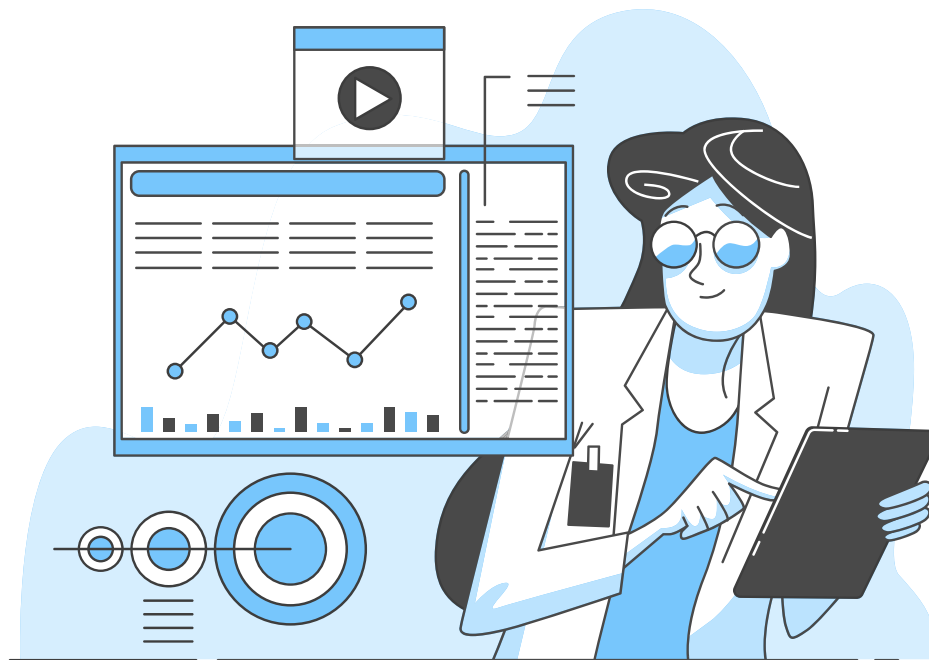
...

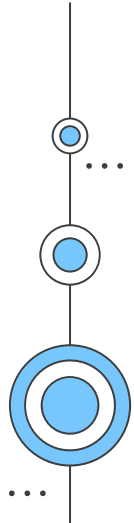
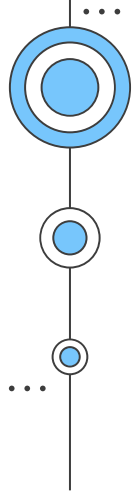
Classe, método, objeto

03

...

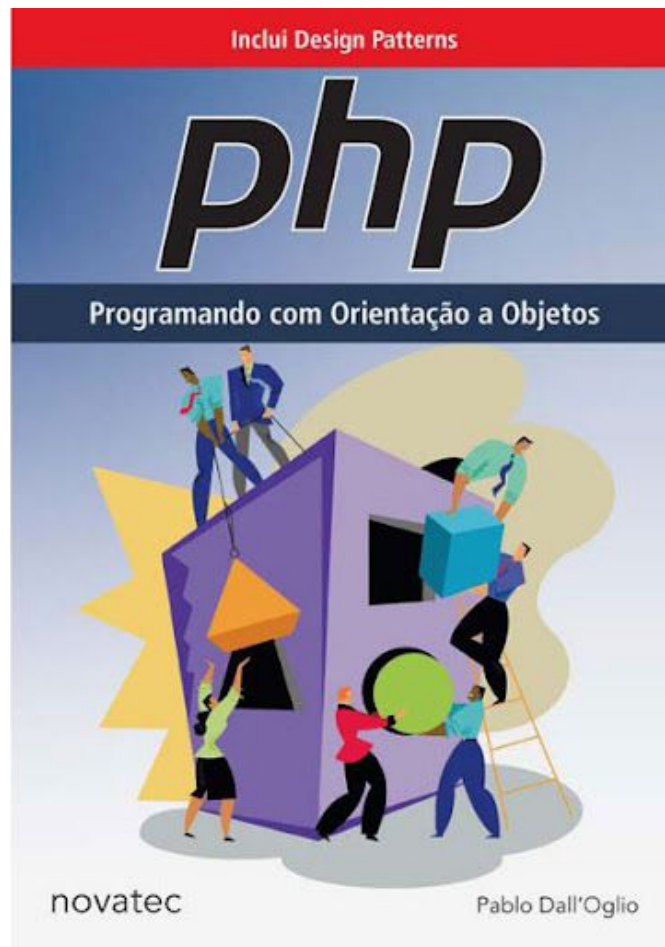
trait, extensão, classe  
abstrata, polimorfismo,  
interface





# Afinal, o que é POO?c

- **Programação orientada a objetos** é um estilo de programação que permite os desenvolvedores agruparem tarefas semelhantes em **classes**.
- Isso ajuda a mantermo-nos dentro do princípio "**don't repeat yourself**"
- Facilitar a manutenção do código.



# DRY (Don't repeat yourself)

- Um dos benefícios é se alguma informação é alterada em seu programa, geralmente, só uma mudança é necessária para atualizar o código.
- O maior mal do programador é manter códigos onde os dados são declarados e redeclarados, acabando num jogo de pique esconde, em busca de funcionalidades e dados duplicados pelo código.



Dando um passo para  
trás para dar dois à  
frente.

Um pouco de Teoria

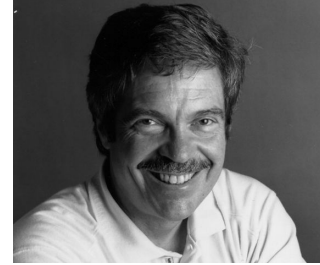


# Como era a vida antes do POO?

- Láaaaa da década de 40, 50 início dos computadores, programação era feita em **baixo nível**.
  - Programação era feita por computadores, da forma que só ele entendia, binário, hexadecimal...
- Surgimento das linguagens de alto nível, **programação linear**.
- **Programação estruturada**, pequenos pedaços de programação linear. (origem dos sistemas).
- Programação modular, módulos estruturados, que eram colocados em cápsulas protegidas, que podem compor sistemas grandes (pouca vida)
- **Programação Orientada a Objetos**



# Quem Criou?

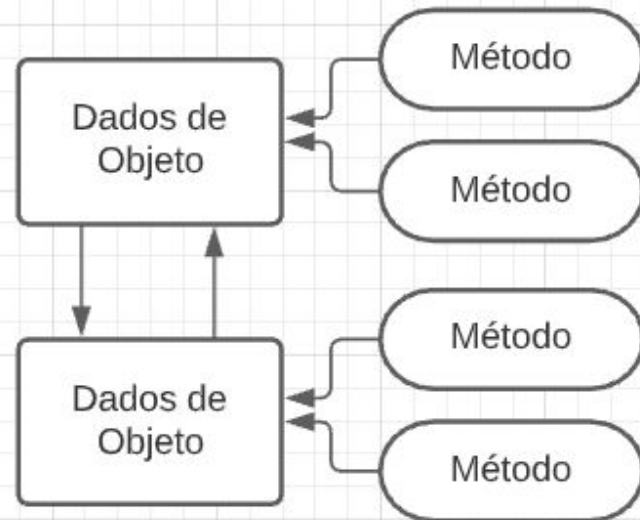
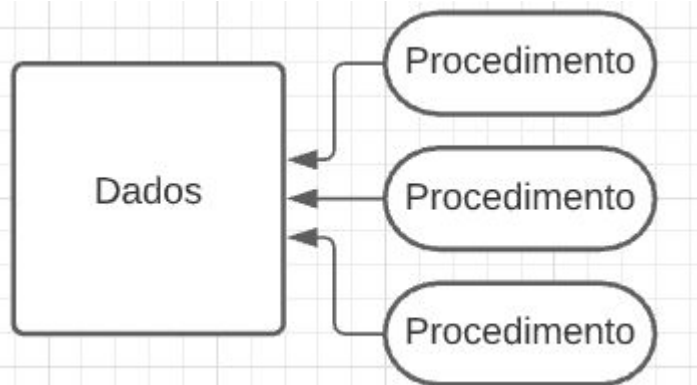


- Alan Kay. (1970)
- Graduado em **matemática** e **Biologia**
- Frase: "O computador ideal deveria funcionar como um organismo vivo, isto é, cada "célula" comportar-se-ia relacionando-se com outras a fim de alcançar um objetivo, contudo, funcionando de forma autônoma.  
As células poderiam também reagrupar-se para resolver um outro problema ou desempenhar outras funções"
- Qualquer coisa que pode ser um objeto, pode ser aplicada em POO.
- Trabalhou na pouco famosa, **Xerox** onde pode ampliar e aplicar seus conhecimentos.
- Trabalhou na Apple, Disney, HP, criou a linguagem smalltalk, TODA em OO.





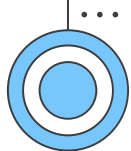
# Como era? Como ficou



# Exemplo Prático

- Vamos explicar na prática usando um **controle**

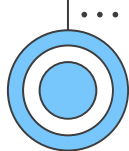




# Linguagens POO

- C++
- Java (Obrigação)
- PHP (Opção)
- Python
- Ruby
- Visual Basic
- Grande maioria das linguagens modernas.





# Tem vantagens? Mas é claro!

- **Confiável** (Isolamento entre as partes gera programas seguros, ao alterar alguma, nenhuma outra é afetada)
- **Oportuno** (Dividir em partes, e desenvolver em partes, programar partes separadamente)
- **Manutenção** (Atualizar é fácil, uma pequena alteração é benéfica para todas as outras)
- **Extensível** (O software não é estático, deve crescer para continuar útil)
- **Reutilizável** (As partes podem ser usadas em outros pontos, classes)
- **Natural** (Uma coisa natural é mais fácil de entender, focar nas features)

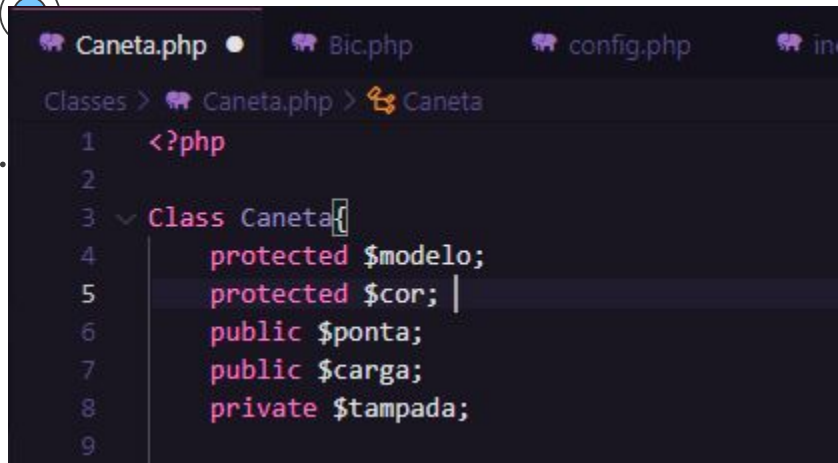
- **Referência: Guanabara, (Curso em Vídeo)**



GALAXPAY  
SEU NEGÓCIO RECOMENDADO



# Visibilidade



```
Caneta.php Bic.php config.php in...
Classes > Caneta.php > Caneta
1  <?php
2
3  Class Caneta{
4      protected $modelo;
5      protected $cor; |
6      public $ponta;
7      public $carga;
8      private $stampada;
9  }
```

— "**public**": Acessível e passível de modificação em qualquer local.

"**protected**": Dentro da classe e das subclasses (herança, ou traits)

"**private**": Apenas dentro da classe que é dona do método/propriedade.

Propriedades e métodos devem ter a visibilidade conforme necessária. Uma dica é iniciar sempre com **private** e mudar para **protected** ou **public** caso **realmente seja necessário**.

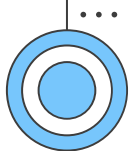


# Encapsulamento

```
<?php
1  <?php
2
3  Class Caneta{
4      protected $modelo;
5      protected $cor;
6      protected $ponta;
7      protected $carga;
8      private $stampada;
9
10
11     protected function getModelo(){
12         return $this->modelo;
13     }
14
15     protected function getCor(){
16         return $this->cor;
17     }
18
19     protected function setModelo(string $modelo){
20         $this->modelo = $modelo;
21     }
22     protected function setCor(string $cor){
23         $this->cor = $cor;
24     }
25 }
26 }
```

- Deixar acessível apenas o necessário provê o encapsulamento.
- Isso se diz respeito a integridade dos dados.
- Ae os dados são passíveis de modificação por qualquer script, não há garantia que os dados permanecem íntegros durante a execução do script
- Não adianta tudo private se tem como acessar tudo do mesmo jeito





# Encapsulamento

- Encapsulamento é mais do que só a visibilidade.
- É garantir a integridade dos dados.

Ou seja:

- só ter método "set" se for necessário
- nesses métodos set, validar o dado, não deixar dar o set se estiver incorreto, etc

```
<?php Untitled-1
1  <?php
2
3  class Caneta{
4      private $modelo;
5
6      public function getModelo(){
7          return $this->modelo;
8      }
9
10     public function setModelo(string $modelo){
11         $this->modelo = $modelo;
12     }
```





**Vamos a Prática!**





# Classes e Objetos

- Classe, é a classificação, idealização, de um objeto, ou grupo de objetos
- Objeto é uma coisa material ou abstrata que pode ser percebida pelos sentidos e descrita por meio de suas **características**, **comportamentos** e **estados** atual. Seria a instância de uma classe.
- Caneta seria uma classe?
- Quais métodos (comportamentos) caneta teria?
- Quais estados uma caneta tem?
- Quais atributos (características) uma caneta tem?



# Classes e Objetos



```
Caneta.php X
Caneta.php > Caneta > rabiscar
1  <?php
2
3  class Caneta{
4      private $modelo;
5      private $cor;
6      private $ponta;
7      private $carga;
8      private $tampada;
9
10     private function rabiscar(){
11         if(!$this->tampada){
12             die("Impossivel rabiscar, caneta tampada");
13         }
14         return "Escrever Rabisco";
15     }
16
17     private function tamparOuDestamparCaneta(bool $status){
18         $this->tampada = $status;
19     }
20
21 }
```





# Como Utilizar?

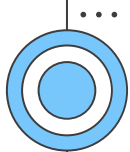
```
8
9 <body>
10   <h1>Aula de POO</h1>
11   <?php
12
13   include 'config.php';|
14
15   $caneta = new Caneta();
16   $caneta->tamparOuDestamparCaneta(false);
17   $rabisco = $caneta->rabiscar();
18
19   var_dump($rabisco);
20
21
22   ?>
23 </body>
24
25 </html>
```



## Aula de POO

Impossível rabiscar, caneta tampada





# Como Utilizar?

```
8
9 <body>
10   <h1>Aula de POO</h1>
11   <?php
12
13     include 'config.php';
14
15     $caneta = new Caneta();
16     $caneta->tamparOuDestamparCaneta(true);
17     $rabisco = $caneta->rabiscar();
18
19     var_dump($rabisco);
20
21   ?>
22 </body>
23
24
25 </html>
```



## Aula de POO

string(16) "Escrever Rabisco"



GALAXPAY  
SEU NEGÓCIO RECORRENTE





Herança



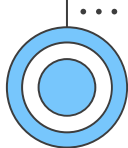
# Herança



```
1  <?php
2
3  Class Bic extends Caneta{
4
5
6      public function __construct()
7      {
8          $this->modelo = "Bic";
9          $this->cor = "Azul";
10         . . .
11     }
12
13 }
```

- “Estender” uma classe, é poder utilizar os métodos, atributos dela em outra classe, “filha” dela.
- Quando algo está ficando específico demais, é necessário ter uma classe específica e as filhas vão utilizando os métodos dela.
- Obs, oq eu tive que alterar para poder ter acesso a esses atributos?

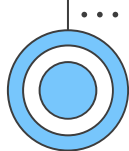




# Herança

- Uma das técnicas para ter **DRY** no código (DON'T REPEAT YOURSELF)
- 
- Deve ser utilizado para agrupar métodos semelhantes, porém com a ideia da classe filha **SER** a classe pai.
- *Exemplo Certo:* Class Pessoa sendo a pai, Class PessoaFisica, Class PessoaJuridica sendo as filhas, PessoaJuridica é Pessoa, então está correto
- *Exemplo Incorreto:* Class Moto sendo a pai, Class Caminhao sendo filho não está correto.  
Pois um caminhão não é uma Moto. O fato dos 2 terem métodos iguais não é o único critério para haver herança.











# Atividades

Chegou momento de organizarmos nossos projetos.

Vamos passar nosso projeto da **locação** para o conceito visto de POO, separado por classes, e fazendo as heranças quando necessário:

Comece separando a classe Connection.php em outras classes específicas para cada escopo diferente que você conseguir identificar.

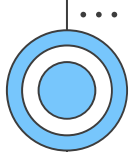
**Uma dica para identificar os escopos:** se seus métodos tiverem sufixos semelhantes, quer dizer que pertencem a um mesmo escopo.

```
>  setAccountToUse  
   getAccount  
>  setUrlClient  
   getUrlClient  
>  getAccountId  
   getCodes
```

No exemplo, temos os escopos "account", "client" e "code"







# Classes Abstratas

Até o momento, a classe pai (ou superclasse) pode ser instanciada normalmente assim como as classes filhas.

Mas em muitos contextos, não faz sentido permitir que isso aconteça. Então podemos declarar a classe pai como `abstract class` para impedir essa ação.

```
<?php

abstract class Connection
{
    // ...
}
```

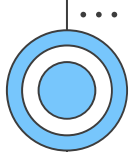
```
<?php

include 'Connection.php';

$conn = new Connection();
var_dump($conn);
```

**Fatal error:** Uncaught Error: Cannot instantiate abstract class Connection in /var/www/html/produtos/webdevs/teste.php:5 Stack trace: #0 {main} thrown in /var/www/html/produtos/webdevs/teste.php on line 5





# Classes Abstratas

As classes abstratas também podem definir métodos abstratos.

Essa declaração apresenta somente a assinatura do método, e deixa a cargo das classes filhas a implementação de acordo com sua regra de negócio.

```
abstract class Connection
{
    abstract public function insert($values);
}
```

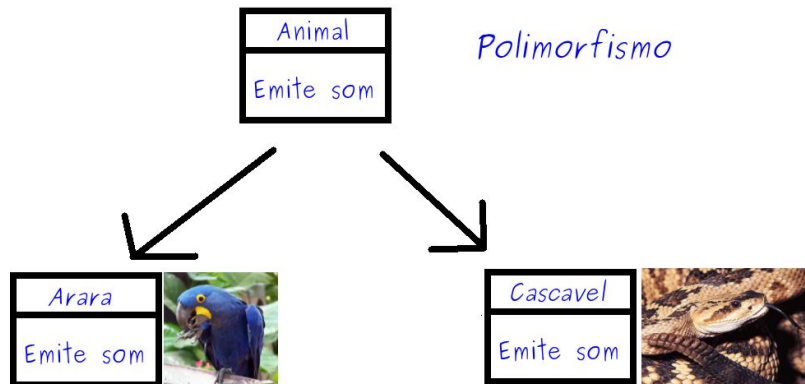
```
class Movie extends Connection
{
    public function insert($values)
    {
        // Aqui você pode fazer validações específicas da classe
        // e escrever o SQL para a tabela
        $sql = 'INSERT INTO filmes (nome_filme, ...) VALUES (...>';
    }
}
```



# Polimorfismo

Na prática, o polimorfismo acontece quando as classes filhas implementam um método que foi definido na classe pai de acordo com suas próprias regras de negócios.

Você pode fazer isso com herança tradicional, ou usando interfaces.



# Traits

Imagine a seguinte situação:

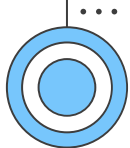
Temos no nosso sistema as entidades Animal e Livro, e ambas possuem data de cadastro.

```
public function setDataCadastro(string $dateFormatted)
{
    $dateObject = \DateTime::createFromFormat('d/m/Y', $dateFormatted);
    $errors = \DateTime::getLastErrors();
    if ($errors['error_count'] > 0) {
        throw new \Exception('data inválida');
    }

    $this->dataCadastro = $dateObject->format('Y-m-d');
}
```

Existe relacionamento de herança entre elas? Como, então, compartilhar o método setDataCadastro() entre as duas, sem repetição de código?





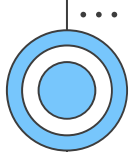
# Traits

O PHP oferece um mecanismo de reutilização de código chamado **Traits**.

Na prática, as traits permitem que o programador reutilize métodos idênticos entre várias classes não-relacionadas, sem a necessidade de criar uma herança entre elas (o que muitas vezes não faria sentido).

No nosso exemplo, como não existe relacionamento de herança entre as classes Animal e Livro, o método setDataCadastro() deveria ficar em uma trait, que será utilizada nas duas classes.





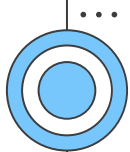
# Traits

```
Animal.php Livros.php HasDataCadastro.php X
base-php > code > landing-page-api > LandingProAPI > Lib > HasDataCadastro.php > {}
1 <?php
2
3 trait HasDataCadastro
4 {
5
6     private $dataCadastro;
7
8     public function setDataCadastro($dateFormatted)
9     {
10         // ...
11     }
12
13 }
14
```

```
Animal.php Livros.php X HasDataCadastro.php
base-php > code > landing-page-api > LandingProAPI > Lib > Livros.php
1 <?php
2
3 trait Livros
4 {
5     use HasDataCadastro;
6
7 }
8
```

```
Animal.php X Livros.php HasDataCadastro.php
base-php > code > landing-page-api > LandingProAPI > Lib > Animal.php >
1 <?php
2
3 trait Animal
4 {
5     use HasDataCadastro;
6
7 }
8
```





# Atividades

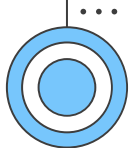
Utilizando o projeto locadora, vamos fazer com o SELECT na tabela filmes retorne um array de objetos, ao invés de um array de arrays.

Dentro da pasta Classes, crie uma nova pasta chamada Mappers. Dentro da pasta Mappers, crie a trait HasDynamicProperty.

```
<?php

trait HasDynamicProperty
{
    public function createProperty($name, $value)
    {
        $this->{$name} = $value;
    }
}
```





# Atividades

Implemente agora a classe Movie. Cada campo da tabela deverá ser convertido em um atributo da classe.

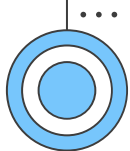
```
<?php

class Movie
{
    use HasDynamicProperty;

    public function __construct(array $dbRow)
    {
        foreach ($dbRow as $name => $value) {
            $this->createProperty($name, $value);
        }
    }
}
```





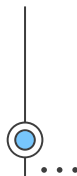


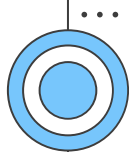
# Atividades

No SELECT na tabela filmes, faça um loop pelos resultados para transformar cada linha em um objeto da classe Movie.

Lembre-se de consertar nos outros arquivos também, para usarem os atributos da classe ao invés das posições do array.

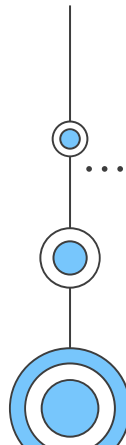
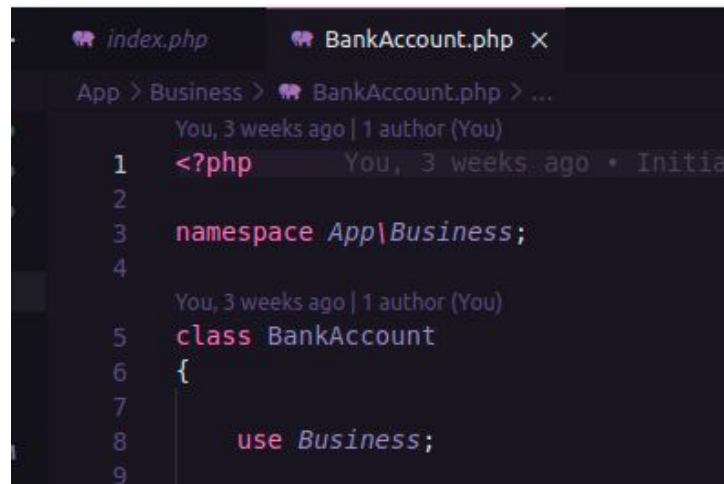
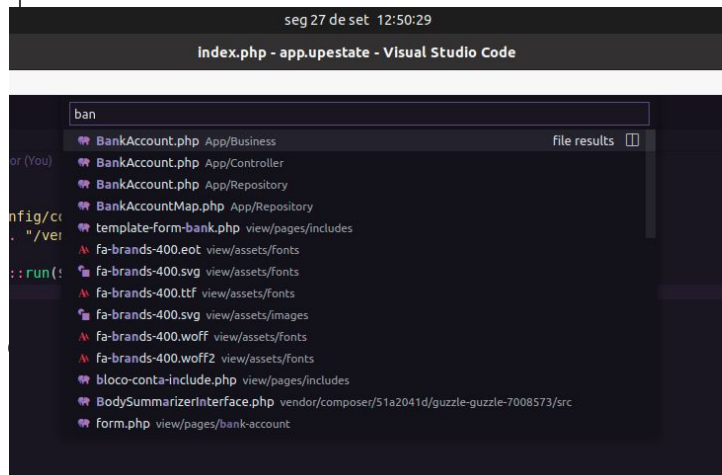
```
public function getMovies()  
{  
    $con = $this->con;  
    $rs = $con->prepare("SELECT * FROM filmes");  
    if ($rs->execute()) {  
        $movies = [];  
        $rows = $rs->fetchAll(PDO::FETCH_ASSOC);  
        foreach ($rows as $row) {  
            $movies[] = new Movie($row);  
        }  
  
        return $movies;  
    }  
    return [];  
}
```

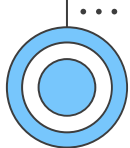




# Namespaces

- Servem para “dividir” as classes em vários “pacotes”
- Podemos repetir o mesmo nome de classe em namespaces diferentes
- Possibilita inclusive o uso mais simples de pacotes externos (ex: composer)
- Pela PSR as pastas devem seguir os nomes do namespace (exceto o primeiro nome, que pode dar uma diferenciada)





# Interfaces

- São utilizadas para normalizar métodos que classes devem ter para que quem for usar a classe não tenha que se preocupar com os detalhes de implementação dela.

Exemplo:

Temos um sistema de e-commerce que disponibiliza várias opções de frete para o cliente escolher na frente da loja: Correios, Jadlog, Azul Express, Melhor Envio.

O cliente **não se importa** em como é feita a implementação do cálculo de frete (se é com API ou webservice, se usa JSON ou XML).

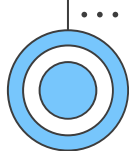
Ele só precisa que as classes tenham os métodos `calcular()`, `getValor()` e `getPrazoEntrega()`.





**Importantes**





# Responsabilidade Única

- Cada classe deve cuidar apenas do seu escopo, ou seja, da sua responsabilidade, assim como cada método.
- Exemplo: uma trait deve adicionar um comportamento e não vários.
- Uma interface deve formalizar um contrato e não vários.
- Indicadores de estar fazendo isso errado:
  - Dificuldade de dar nomes.
  - Muitas linhas de código.
  - Muitos ifs aninhados.
  - Dificuldade em dar manutenção



# Atividades

- 1) Fazer um Script para cálculo de frete.
- Estrutura:  
App
  - Frete
    - TNT (1.33% do produto)
    - JADLOG (2% do prd - 11)
    - CORREIOS ( $\text{Raiz de } 16 * 10 \% \text{ do produto} + 1,70$ )

Se o valor vier negativo, o valor do frete será igual a metade do valor do produto.

