

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Orientação a Objetos no Delphi for PHP

Como aplicar conceitos de POO em aplicações PHP — Parte 2

Como prometido em meu primeiro artigo vamos dar continuidade a este assunto tão extenso que é a orientação a Objetos. Como dito anteriormente toda linguagem dita OO deve estar apoiada nos pilares desta filosofia que são: *Herança*, *Encapsulamento*, *Abstração* e *Polimorfismo* e o PHP 5 implementa essas quatro características inclusive o *polimorfismo* que é o foco do nosso artigo.

Muitos programadores vêem a herança como a solução de seus problemas, porém se usada indiscriminadamente ele pode ser um feitiço que se vira contra o feiticeiro, isto porque se não tomarmos os devidos cuidado acabamos por acoplar todo o nosso código através da herança. Porém se há algo que justifique o uso da herança este se chama *polimorfismo*.

O *polimorfismo* é um dos assuntos mais importantes na POO. Com o uso de classes e heranças conseguimos facilmente descrever uma situação da vida real além

de conseguir estender projetos por meios da reutilização de código. Mas se de um lado o *polimorfismo* é um dos assuntos mais importantes da POO é também sem dúvida nenhuma um dos mais difíceis de ser compreendido. Isto porque muitos que se aventuram pela primeira vez na OO entram ainda com uma mentalidade *procedural* e enxergando as classes como entidades do banco de dados tornando mais difícil a compreensão da POO como um todo.

Polimorfismo

Teoricamente e a grosso modo entende-se por *polimorfismo* “a capacidade que um mesmo método tem de se comportar de maneira diferente dependendo de qual classe ele foi chamado”. Nós conseguimos isso declarando um método em uma classe base e o sobrescrevemos em uma classe herdada, facilitando assim o desenvolvimento e reaproveitamento códigos.

Antes de mais nada é importante dei-



Rodrigo Carreiro Mourão

(rodrigocarreiro@tdstecnologia.com.br)

Consultor da TDS Tecnologia RJ atuando na área de desenvolvimento de projetos Orientados a Objetos, Design Patterns, MVC. BDS2006 Win32 Product Certified. Borland Instructor Certified. Instrutor de treinamento oficiais CodeGear DelphiWin32, Delphi for PHP, Delphi .Net. Palestrante da Borland Conference 2007.

xar claro que a linguagem que estamos trabalhando é o PHP e que o mesmo é interpretado, fracamente *tipado* e sendo assim não há a necessidade de se declarar variáveis com seus tipos. Mas o que isso tem a ver com *polimorfismo*? Tudo! Observe o código da **Listagem 1**.

No código da **Listagem 1** será levantada uma exceção no evento *OnClick* do botão com a mensagem “Nem todo ser vivo produz som!!!” mesmo um objeto do tipo *THomem* sendo instanciado para a variável *Ser*. Isso acontece porque no Delphi os métodos são estáticos por padrão. Isso significa que independente do objeto que seja instanciado na variável vale para ela o tipo que foi declarado. Para conseguirmos o efeito desejado, ou seja, o método *ProduzirSom* da classe *THomem* seja chamado, é preciso declarar o método *ProduzirSom* na classe *TSerVivo* como *Virtual*.

Isso faz com que o Delphi crie uma VMT (“Virtual Method Table”) uma tabela com os ponteiros dos métodos que foram declarados como virtuais para que possam ser sobrescritos em tempo de execução. Na **Listagem 2** vemos como ficaria o código.

No PHP nada disso é necessário, como ele e fracamente tipado, ou seja, não

Listagem 1. Polimorfismo no Delphi Win32

```
unit SeresVivos;

interface

type
  TServivo = class
    procedure ProduzirSom;
  end;

  TCachorro = class(TServivo)
    procedure ProduzirSom;
  end;

  THomem = class(TServivo)
    procedure ProduzirSom;
  end;

implementation

uses
  SysUtils, Dialogs;

{ TCachorro }

procedure TCachorro.ProduzirSom;
begin
  ShowMessage('Cachorro Latindo !!!');
end;

{ TServivo }

procedure TServivo.ProduzirSom;
begin
  raise Exception.Create('Nem todo ser vivo produz som !');
end;

{ THomem }

procedure THomem.ProduzirSom;
begin
  ShowMessage('Homem Falando !!!');
end;

end.

procedure TForm1.Button1Click(Sender: TObject);
var
  Ser: TServivo;
begin
  Ser := THomem.Create;
  Ser.ProduzirSom;
end;
```

Listagem 2. Métodos Virtuais

```
unit SeresVivos;

interface

type
  TServivo = class
    procedure ProduzirSom; virtual;
  end;

  TCachorro = class(TServivo)
    procedure ProduzirSom; override;
  end;

  THomem = class(TServivo)
    procedure ProduzirSom; override;
  end;

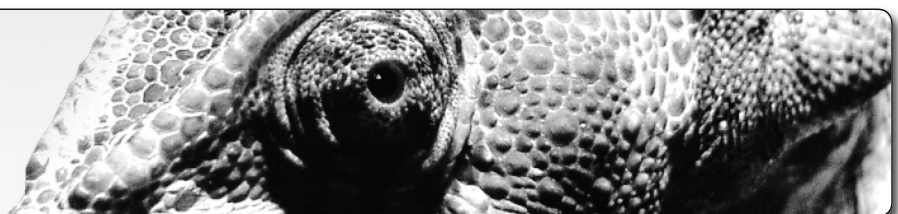
implementation

end.
```



Nota do DevMan

O termo Programação Procedural (ou programação procedimental) é às vezes utilizado como sinônimo de Programação Imperativa (paradigma de programação que especifica os passos que um programa deve seguir para alcançar um estado desejado), mas o termo pode se referir (como neste artigo) a um paradigma de programação baseado no conceito de chamadas a procedimentos. Procedimentos, também conhecidos como rotinas, sub-rotinas, métodos, ou funções simplesmente contêm um conjunto de passos computacionais a serem executados. Um dado procedimento pode ser chamado a qualquer hora durante a execução de um programa, inclusive por outros procedimentos ou por si mesmo.



precisamos declarar o tipo de nossas variáveis, não há porque os métodos serem estáticos, e nem podem. Como não há um tipo definido para o objeto que será instanciado, por padrão será chamado o método do objeto que foi instanciado para aquela variável. Por isso podemos dizer que no PHP os métodos são “Virtuais” e isto facilita a implementação do *polimorfismo* reduzindo assim o uso de controles de fluxos como *If's* para tornar nosso código mais robusto e expansível.

Você deve estar se perguntando: “Se não precisamos declarar o tipo de nossas variáveis e de nossos objetos, como saber se um objeto pertence a uma determinada hierarquia ou possui o método que queremos chamar?”.

Para isto temos no PHP 5 o operador *instanceof*. Ele verifica se um objeto qualquer possui um relacionamento *é um* com uma classe específica. Ele na verdade é uma modificação da já existente função *is_a()* que foi descontinuada, pois o primeiro é um operador binário lógico permitindo criar sentenças como a seguir:

```
if($c instanceof Cliente) {  
    echo 'c é um cliente';  
}
```

Para exemplificar essa facilidade observe o código da **Listagem 3**.

Podemos notar facilmente que o código escrito não está expansível. Imagine que você decida criar em seu modelo mais cinco animais cada um produzindo seu respectivo som, isso se tornaria para você um tormento, pois teria que adicionar outros cinco blocos *elseif* para fazer a verificação. Isso fere o conceito de reaproveitamento de código e pior do que isso imagine o sofrimento que seria para dar manutenção em um código deste porte. Então esse código está totalmente fora de cogitação.

O uso da herança com o *polimorfismo* nos ajuda a resolver este problema, a intenção é fazer a mesma coisa, porém de uma maneira que nos ajude a expandir nosso código, reaproveitá-lo e mais do que isso que facilite a manutenção do mesmo. Isso aplicado na prática se torna o código descrito na **Listagem 4**.

Note que agora criamos uma classe *SerVivo* que servirá de base para todas as outras classes de animais que iremos criar e que irão produzir algum som. É nesta classe que está declarado o método que será sobrescrito nas classes descendentes porém não há aqui a necessidade de nenhum parâmetro ou diretiva para tornar este método virtual e passível de ser sobrescrito. Observe que desta maneira o nosso código não fica “engessado”. Se por exemplo decidirmos criar uma classe *Gato* em nada teremos que alterar o método *GetSom*. Basta que a classe *Gato* herde de *SerVivo* para que seja aceita no método *GetSom* e feito isso sobrescrevemos o método *ProduzirSom* na classe herdada.

Perceba que foi utilizado em nosso código o operador *instanceof* para verificar se o objeto passado como parâmetro para o método *GetSom* pertence à classe *SerVivo* ou herda da mesma. Se esta condição for satisfeita podemos ter a certeza que ele terá em sua estrutura o método *ProduzirSom*. Agora qual mensagem será exibida por este método vai depender do tipo do objeto que estiver instanciado ali no momento. Com isso voltamos à definição do início do nosso artigo: “*Polimorfismo: a capacidade que um mesmo método tem de se comportar de maneira diferente dependendo de qual classe ele foi chamado*”.

Claro que muito mais pode ser feito para tornar nosso código mais coeso, por exemplo, se nem todos os seres vivos produzem som não há porque o método *ProduzirSom* na classe *SerVivo* tenha implementação. Neste caso aplica-se um método abstrato.

Métodos e Classes Abstratas

No PHP5 temos também o conceito de classes e métodos abstratos. Diz abstrato um método que não possui implementação ou uma classe que não deve ser instanciada, que serve apenas de referência para ser herdada e o método para ser sobrescrito.

Isto é muito comum em OO, pois quando trabalhamos com grandes equipes e em grandes projetos geralmente queremos garantir que determinadas classes tenham certos métodos essenciais ao funcionamento do modelo e como são vários os

programadores envolvidos no processo definimos esses métodos como abstratos demonstrando assim a nossa intenção de que ele seja sobrescrito na classe que o herdou. Com isso temos um código mais organizado, mais coeso. A própria classe nos lembra que precisamos implementar esse método em algum momento. No BDS 2006, quando temos um método abstrato em uma classe e herdamos da mesma, ao utilizarmos o *CodeCompletion* este método aparece em vermelho indicando que precisa ser implementado.

Um fato importante a ser lembrado é que, no PHP se uma classe possui ao menos um método abstrato ela deve ser declarada como abstrata diferente de outras linguagens de programação. Essa dupla definição existe para permitir a você declarar uma classe como abstrata mesmo que a mesma não possua métodos abstratos.

No exemplo dos nossos seres vivos fica claro que nem todos os seres produzem som, assim este método nesta classe não deve ter implementação ficando nossa classe como a da **Listagem 5**.

Note na **Listagem 5** que a classe *SerVivo* agora foi declarada como *abstract*, pois dos três métodos que a mesma possui um deles é abstrato e por isso temos que declarar a classe como abstrata também. Assim *SerVivo* não poderá ser instanciada e também não haverá necessidade, pois esta classe é apenas uma base para as demais.

Como todo ser vivo nasce, cresce, se reproduz e morre, podemos observar que temos algumas dessas fases em forma de métodos em *SerVivo*, não como abstratos, mas passíveis de serem sobrescritos.

Métodos e Propriedades Estáticas

É de conhecimento geral que uma classe pode declarar “n” propriedades e métodos. Cada instância desta classe, ou seja, cada objeto possui uma cópia destas propriedades e métodos. Isso significa que podem receber valores distintos e seus métodos serem chamados de forma independente. Porém em OO temos um conceito chamado de propriedades estáticas e métodos estáticos. São elementos que pertencem à classe e não ao objeto, com isto eles podem ser acessados e invocados a partir da classe sem a necessidade de se instanciar o objeto.

Podemos ter vários objetos, mas todos eles oriundos de uma única classe. Como as propriedades e métodos pertencem à classe podemos deduzir que essas propriedades e métodos são compartilhados por todos os objetos.

Há algumas considerações sobre o uso de métodos e propriedades estáticas no PHP. Por exemplo, para se acessar propriedades estáticas de dentro da própria classe temos que utilizar o operador *Self::* que aqui tem uma função oposta a do Delphi onde o *Self* é usado para fazer referência ao objeto do contexto enquanto no PHP usamos para acessar a classe. Outro detalhe é que para acessar uma propriedade ou método deste tipo temos que fazê-lo através da classe de duas maneiras:

```
Classe::Propriedade
Classe::Método()
```

Mas você deve estar se perguntando se realmente fará uso deste tipo de recurso e se ele é realmente necessário. Pois então observe o código da **Listagem 6**.

O que temos aqui é uma classe cliente comum e simples, porém com um contador que registra a quantidade de objetos desta classe que foram instanciados. Por isso a propriedade estática *\$Contador*, propriedade esta que pertence à classe e não ao objeto, é compartilhada por todas as instâncias. Observe que no construtor acessamos esta propriedade através do operador *Self* e incrementamos em um. Feito isso atribuímos o valor do contador à propriedade *\$id*, essa sim pertencente ao objeto.

O método *CriarClientes* executa um laço *for* e instancia dez objetos *Clientes* e logo após mostra no navegador o valor da propriedade *\$id*.

Poderíamos incluir nesta classe *Cliente*

um método para obter o valor atual do contador da classe. Este método deverá ser estático para evitar que se tenha que instanciar a classe para invocar o método. Veja como ficaria com essa modificação na **Listagem 7**.

Observe que adicionamos a nossa classe *Cliente* um método estático que retorna a posição atual do contador. Para invocá-lo utilizamos a própria classe sem a necessidade de se instanciar um objeto: *Cliente::GetCount()*.

Criando um exemplo

É notório que os exemplos citados acima servem apenas para exemplificar os conceitos abordados aqui. Para que você possa ter uma noção de como o polimorfismo auxilia em tarefas diárias vamos construir um pequeno exemplo abordando o conceito acima. Abra o Delphi for

Listagem 3. Código PHP sem polimorfismo

```
<?php
class Cachorro{
    function Latir(){
        echo "Cachorro Latindo !";
    }
}
class Homem{
    function Falar{
        echo "Homem Falando !";
    }
}
function ProduzirSom($Obj){
    if ($Obj instanceof Cachorro){
        $Obj->Latir();
    }elseif($Obj instanceof Homem){
        $Obj->Falar();
    }else{
        echo "O objeto passado não produz som";
    }
}
?>
```

Listagem 4. Polimorfismo aplicado no PHP

```
<?php
class SerVivo{
    function ProduzirSom(){
        echo "Nem todo ser vivo produz som !";
    }
}
class Cachorro extends SerVivo{
    function ProduzirSom(){
        echo "Cachorro Latindo !";
    }
}
class Homem extends SerVivo{
    function ProduzirSom{
        echo "Homem Falando !";
    }
}
function GetSom($Obj){
    if ($Obj instanceof SerVivo){
        $Obj->ProduzirSom();
    }else{
        echo "O objeto passado não é um ser vivo !";
    }
}
?>
```

Listagem 5. Classes abstratas no PHP

```
<?php
abstract class SerVivo{
    function Nascer(){
        echo "Nascendo !!!";
    }
    function Crescer(){
        echo "Crescendo !!!";
    }
    abstract function ProduzirSom();
}

class Cachorro extends SerVivo{
    function ProduzirSom(){
        echo "Cachorro Latindo !";
    }
}

class Homem extends SerVivo{
    function ProduzirSom{
        echo "Homem Falando !";
    }
}
function GetSom($Obj){
    if ($Obj instanceof SerVivo){
        $Obj->ProduzirSom();
    }else{
        echo "O objeto passado não é um ser vivo !";
    }
}
?>
```

Listagem 6. Propriedade Estática no PHP

```
<?php
class Cliente{
    static $Contador = 0;
    public $id;
    function __construct(){
        self::$Contador++;
        $this->id = self::$Contador;
    }
    function CriarClientes(){
        for($i=0;$i<=9;$i++){
            $cliente = new Cliente();
            echo $cliente->id."<br>";
        }
    }
}
?>
```

PHP e crie uma nova aplicação. Para isso acesse o menu *File|New>Application*.

Salve a aplicação pressionando *Ctrl+Shift+S* ou escolha *File>Save Project As* com o nome "PrjPolimorfismo.php" e em seguida salve a *Unit* como "index.php". Altere a propriedade *Name* do formulário para "FrmIndex". Este formulário será utilizado para criar a interface visual do nosso exemplo.

Adicione neste formulário três *Edits* da paleta *Standard*. Altere seus nomes para "EdtNome", "EdtEmail", "EdtRegistro", respectivamente. Adicione três *Labels*, também da paleta *Standard*, um para cada *Edit* e troque seus *Captions*

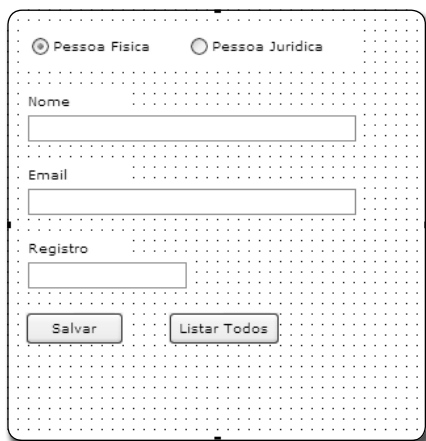


Figura 1. Exemplo de layout

de acordo com cada campo. Adicione também dois Buttons ("BtnSalvar" e "BtnListar") e um *RadioGroup*. Adicione ao *RadioGroup* dois itens a sua propriedade *Items*. Digite "Pessoa Física" e "Pessoa Jurídica". Altere a propriedade *Orientation* para *orHorizontal*. Na **Figura 1** temos uma sugestão de *layout*.

Concluindo o layout passaremos agora para a criação de nossas classes. Não irei me aprofundar na sintaxe OO no PHP pois este foi escopo do primeiro artigo.

Acesse o menu *File|New>Unit* e crie uma nova *Unit* salvando-a com o nome "Model.php". Nesta *Unit* iremos criar um modelo onde teremos uma classe *Pessoa* com as propriedades *Nome* e *Telefone* comum a todo o tipo de pessoa. Faremos uma especialização de *Pessoa* em *PessoaFisica* e *PessoaJuridica* cada uma com uma propriedade exclusiva *CPF* e *CNPJ* respectivamente. Na *Unit* criada digite o código da **Listagem 8**.

No código da **Listagem 8** apenas criamos a estrutura das classes que serão usadas em nosso exemplo. Posicione o cursor na área da classe *Pessoa* e pressione *Ctrl + Shift + Alt + U* para publicar as propriedades. No diálogo que se abre preencha os campos conforme a **Figura 2**.

Repare que o Delphi for PHP já adiciona

à classe um campo protegido *\$_nome* com os métodos *Get* e *Set* para acesso a este *Field*. Isso fará que no *Code Completion* você possa acessar a propriedade *Nome* através de *Pessoa->Nome* embora o *GetNome* e o *SetNome* não sejam invocados.

Retornando ao exemplo, repita o passo anterior e faça o mesmo para publicar em *Pessoa* a propriedade *Telefone*. Em *PessoaFisica* publique a propriedade *CPF* e em *PessoaJuridica* a propriedade *CNPJ*. Observe na **Listagem 10** as classes com as propriedades publicadas.

A princípio você deverá notar que não foge muito a regra de criação de classe em comparação com o Delphi Win32. Temos em *Pessoa* as propriedades pertinentes a todo tipo de pessoa. *PessoaFisica* herda de *Pessoa*, e por esse motivo traz consigo, devido a herança, as propriedades *Nome* e *Telefone*. Precisamos apenas adicionar a propriedade *CPF* assim como acontece em *PessoaJuridica*, sendo que nesta temos o *CNPJ* como diferencial.

Se notou bem, publicamos o método *Save()* em *Pessoa* como *abstract* indicando assim a intenção que temos em sobrescrevê-lo nas classes descendentes e é isso que faremos agora. Na classe *PessoaFisica* sobrescreva a função *Save()* conforme a **Listagem 11**. Faça o mesmo para *PessoaFisica* conforme **Listagem 12**.

Listagem 7. Método Estático no PHP

```
<?php
class Cliente{
    static $Contador = 0;
    public $id;
    function __construct(){
        self::$Contador++;
        $this->id = self::$Contador;
    }
    static function GetCount(){
        return self::$Contador;
    }
}
function CriarClientes(){
    for($i=0;$i<=9;$i++){
        $cliente = new Cliente();
        echo $cliente->id."<br>";
    }
    echo Cliente::GetCount();
}
?>
```

Listagem 8. Estrutura das classes

```
abstract class Pessoa{
    protected abstract function Save();
}
class PessoaFisica extends Pessoa{
}
class PessoaJuridica extends Pessoa{
}
```



Nota do DevMan

Para que os métodos *Get* e *Set* possam ser invocados automaticamente como no Delphi Win32 suas classes precisam possuir os métodos *__set()* e *__get()* ou simplesmente herdar suas classes da classe *Object* do Delphi for PHP o que não acontece automaticamente como no caso do Delphi Win32. A classe *Object* já intercepta o acesso a propriedade e delega a chamada aos respectivos *get's* e *set's*. Observe na **Listagem 9** a estrutura em *Object*.

Declaramos os métodos *__get()* e *__set()* e preparamos ambos para que localizem a herança e retornem o resultado. Caso não seja encontrada, uma exceção é levantada informando que o método não existe.

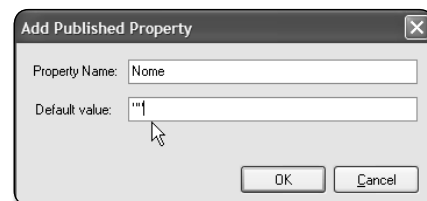


Figura 2. Publicando a Propriedade Nome

Esses são os métodos que serão fruto do *polimorfismo*. Teremos uma outra classe que servirá como lista de pessoas, onde guardaremos todos os objetos criados na aplicação. O botão listar servirá para percorrer toda a lista e invocar o método *Save()* dos objetos contidos nela. Não sabemos ao certo se são pessoas físicas ou jurídicas. Apenas sabemos que são pessoas e se são pessoas possuem o método *Save()*. Na mesma *Unit* do modelo, *model.php*, crie a classe *ListaPessoa* conforme **Listagem 13**.

Nesta classe temos um *Array Private* onde guardaremos todos os objetos criados na aplicação. O método público *AddPessoa* serve para que possamos ter uma maneira de adicionarmos itens ao *Array* que é privado. Repare no parâmetro da função que colocamos o tipo do parâmetro antes do mesmo. O PHP não é tipado, porém temos este recurso que é chamado de *Hint de Classe*.

Com isso um erro será gerado se tentarmos adicionar no *Array* um objeto que não seja do tipo pessoa. A função publicar em *ListaPessoa* apenas faz um *loop* no *Array* onde cada interação, ou seja, cada vez que passa por um objeto o método *Save()* é invocado, publicando no *browser* as informações do objeto.

Para testar nosso modelo vamos ao evento *OnClick* do botão *Salvar*. Neste

evento digite o código da **Listagem 14**.

O procedimento é simples. Primeiro verificamos a opção selecionada no *RadioGroup* através da propriedade *ItemIndex*. Feito isso criamos o objeto de acordo com a opção do usuário e já carregamos o valor do *edtRegistro* para a propriedade *CPF* ou *CNPJ* dependendo do caso.

Em seguida carregamos para as propriedades *Nome* e *Email* os valores de seus respectivos *Edit's*. Por fim verificamos se há na seção uma variável *Obj*. Se ela não existir criamos um objeto e o colocamos na seção para poder passá-lo para outra página. Se a variável de seção já existir então apenas invocamos de dentro da seção a função *AddPessoa* passando o objeto criado. Ao final teremos na seção um objeto *ListaPessoa* com todas as pessoas criadas a cada clique de botão.

Para concluir nosso exemplo no *OnClick* do botão *Listar Todos* chamaremos uma página ("lista.php") e nela exibiremos o conteúdo de nossa lista de pessoas. Para isso codifique o botão em questão fazendo uma chamada ao método *redirect* do PHP, conforme a seguir:

```
redirect('lista.php');
```

Esta página ainda não foi criada então proceda com a criação em *File|New>Form*. Salve-a como "lista.php" e modifique

Listagem 9. Métodos *__get* e *__set* em Object.

```
<?php
function __get($nm){
    $method='get'.$nm;
    if (method_exists($this,$method)){
        return ($this->$method());
    }else{
        $method='read'.$nm;
        if (method_exists($this,$method)){
            return ($this->$method());
        }else{
            if ($this->inheritsFrom('Component')){
                if( isset($this->_childnames[$nm]) )
                    return $this->_childnames[$nm];
            }
            throw new EPropertyNotFound(
                $this->ClassName()." ".$nm);
        }
    }
}

function __set($nm, $val){
    $method='set'.$nm;
    if (method_exists($this,$method)){
        $this->$method($val);
    }else{
        $method='write'.$nm;
        if (method_exists($this,$method)){
            $this->$method($val);
        }else{
            throw new
            EPropertyNotFound($this->
                ClassName()." ".$nm);
        }
    }
}
?>
```

Listagem 10. Classes com propriedades publicadas

```
<?php
/* Classe principal Pessoa */
abstract class Pessoa{
    protected $_nome="";
    function getNome(){
        return $this->_nome;
    }
    function setNome($value){
        $this->_nome=$value;
    }
    function defaultNome(){
        return;
    }
    protected $_email="";
    function getEmail(){
        return $this->_email;
    }
    function setEmail($value){
        $this->_email=$value;
    }
    function defaultEmail(){
        return;
    }
    protected abstract function Save();
}

/* Classe PessoaFisica */
class PessoaFisica extends Pessoa{
    protected $_cpf="";
    function getCPF(){
        return $this->_cpf;
    }
    function setCPF($value){
        $this->_cpf=$value;
    }
    function defaultCPF(){
        return;
    }
}

/* Classe PessoaJuridica */
class PessoaJuridica extends Pessoa{
    protected $_cnpj="";
    function getCNPJ(){
        return $this->_cnpj;
    }
    function setCNPJ($value){
        $this->_cnpj=$value;
    }
    function defaultCNPJ(){
        return;
    }
}
?>
```

Figura 3. Cadastrando Pessoas na Lista

Listagem 11. Método Save() na classe PessoaFisica

```
public function Save(){
    echo '<font size=6 color=darkblue'.
        'face=Tahoma'. $this->Nome. '</font><br>'.
        'Email...: '. $this->Email. '<br>'.
        'CPF...: '. $this->CPF. '<br>'.
        'Tipo...: Pessoa Fisica <br>';
}
```

Listagem 12. Método Save() na classe PessoaJuridica

```
public function Save(){
    echo '<font size=6 color=darkblue'.
        'face=Tahoma'. $this->Nome. '</font><br>'.
        'Email...: '. $this->Email. '<br>'.
        'CNPJ...: '. $this->CNPJ. '<br>'.
        'Tipo...: Pessoa Juridica <br>';
}
```

Listagem 13. Classe Pessoa Lista

```
class ListaPessoa{
    static private $instance;
    private $_list = array();

    function AddPessoa(Pessoa $Obj){
        $this->_list[] = $Obj;
    }

    function Publicar(){
        for ($i = 0; $i < count($this->_list); $i++){
            $this->_list[$i]->Save();
        }
    }
}
```

Listagem 14. Criando os objetos do modelo

```
function BtnSalvarClick($sender, $params){
    switch ($this->RadioGroup1->ItemIndex) {
        case 0:
            $Pessoa = new PessoaFisica;
            $Pessoa->CPF = $this->Edit3->Text;
            break;
        case 1:
            $Pessoa = new PessoaJuridica;
            $Pessoa->CNPJ = $this->Edit3->Text;
    }

    $Pessoa->Nome = $this->Edit1->Text;
    $Pessoa->Email = $this->Edit2->Text;

    if (!isset($_SESSION['Obj'])){
        $_SESSION['Obj'] = new ListaPessoa;
    }

    $_SESSION['Obj']->AddPessoa($Pessoa);
}
```

seu *Name* para "FrmLista". No *OnShow* dessa página digite o código a seguir:

```
$L = $_SESSION['Obj'];
$L->Publicar();
```

Este código apenas invoca o método *Publicar* do objeto *ListaPessoa* que está na seção. Este método por sua vez faz um *loop* no *Array* invocando o método *Save()* de cada item da lista. Com isso quando esta página for carregada exibirá no *browser* todas as pessoas cadastradas na lista devidamente formatadas.

Execute a aplicação. *Selecione* o tipo de pessoa, informe os dados e clique no botão *Salvar*. Veja a aplicação em execução nas **Figuras 3 e 4**.

Conclusão

Com os conceitos aprendidos até o momento já é possível adicionar aos nossos projetos um aspecto mais profissional, e claro, facilitar bastante o nosso trabalho. Confesso que para quem não tem um certo conhecimento em OO pode parecer a primeira vista um tanto complicado aplicar todos esses conceitos, mas com o tempo você irá perceber que as coisas não são tão complicadas como parecem. O mundo de programação OO é fascinante e requer muita dedicação por parte daqueles que se lançam nele. Por isso dedique-se. Lembre-se que OO é OO independente da linguagem. Habitue-se a pensar OO, pois todas as grandes linguagens e por consequência ferramentas estão apoiadas nesta filosofia.

Um grande abraço a todos e até o próximo artigo, onde trataremos de um assunto um tanto interessante: Design Patterns em PHP.

Eu sou Rodrigo Carreiro e pela sua atenção muito obrigado. ●

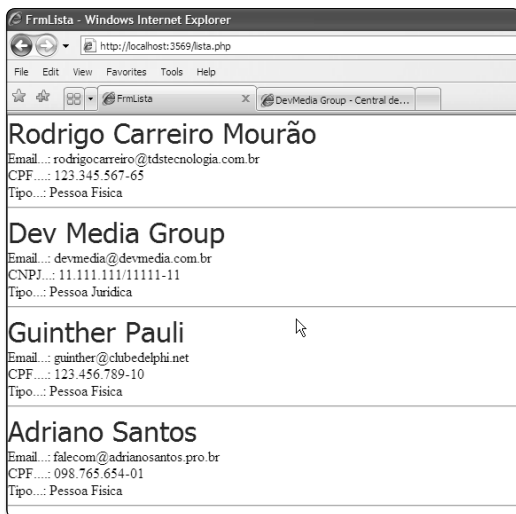


Figura 4. Publicando conteúdo da lista



Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/javamagazine/feedback

