**Classes e Objetos**

Bem vindo ao nosso curso de PHP e Orientação a Objetos. No decorrer deste módulo entenderemos alguns dos principais conceitos e recursos de OO e colocaremos isso em prática na aplicação que foi desenvolvida no curso de PHP e Mysql. Não se preocupe se você não fez o outro curso, no primeiro exercício deste capítulo você receberá instruções para baixar o projeto e configurar o banco para continuar a partir deste ponto.

**Uma forma melhor de representar um Produto**

Da forma que está, as informações de um Produto estão todas espalhadas pelas assinaturas de nossas funções. Você pode notar isso no método insereProduto, do arquivo banco-produto.php:

* 1. function insereProduto($conexao, $nome, $preco, $descricao, $categoria\_id, $usado)
  2. {
  3. if(array\_key\_exists('usado', $\_POST)) {
  4. $usado = "true";
  5. } else {
  6. $usado = "false";
  7. }
  8. $query = "insert into produtos (nome, preco, descricao, categoria\_id, usado)
  9. values ('{$nome}', '{$preco}', '{$descricao}', '{$categoria\_id}', {$usado})";
  10. return mysqli\_query($conexao, $query);
  11. }

Um dos problemas disso é que a cada nova informação que um produto venha a ter, um novo argumento precisa ser passado. Com o passar do tempo, a assinatura da função insereProduto vai ficando ainda maior, com mais argumentos, difícil de ler e cada vez mais difícil de manter.

Ao passar todas as informações do produto espalhadas dessa forma, fica fácil confundir a ordem desses argumentos. Podemos por engano passar o email no lugar do nome e assim por diante.

Uma das primeiras propostas de escrever um código orientado a objetos é que se um elemento é importante em nosso contexto, ele deve ser representado por um tipo próprio. Claramente um Produto é extremamente importante para nossa aplicação de loja, portanto, nada mais justo do que criar um tipo Produto para representá-lo, no lugar de trabalhar com suas informações espalhadas. Nosso código ficará muito mais organizado.

**Criando uma classe em PHP**

Vamos criar o arquivo produto.php, com a definição inicial de um Produto. Isso pode ser feito da seguinte forma:

* 1. <?php
  2. class Produto
  3. {
  4. // ...
  5. }

Repare que a estrutura de uma classe é bem simples, ela começa com a palavra reservada class, seguida de seu nome e um par de chaves. Uma classe é um molde, uma receita de como criar determinado **objeto** de nosso sistema.

**Convenção de nomes para classes**

O ideal é que você sempre dê nomes significativos para suas classes, refletindo bem o que determinado objeto desse tipo representa em seu sistema. Por esse motivo é bastante recomendado que você não use muitas abreviações ao declarar seu nome. Além disso, por padrão, uma classe deve começar com letra maiúscula e ter seu case alterado a cada nova palavra (o famoso ::CamelCase:: ou ::CamelCaps::). Você pode ler mais sobre convenções de código ::PHP:: em:

<http://git.php.net/?p=php-src.git;a=blob_plain;f=CODING_STANDARDS;hb=HEAD>

**Propriedades da classe**

A classe Produto por enquanto está vazia, mas uma classe deve representar tudo o que um objeto desse tipo tem e tudo o que ele faz. Por enquanto, vamos focar no que um Produto tem, que já sabemos ser um nome, preco, etc. Podemos fazer isso da seguinte forma:

* 1. <?php
  2. class Produto
  3. {
  4. public $id;
  5. public $nome;
  6. public $preco;
  7. public $descricao;
  8. public $categoria;
  9. public $usado;
  10. }

Não se intimide com a presença desse novo modificador, o public. Ele e outros modificadores de visibilidade serão melhor detalhados mais à frente. Apesar de se parecer muito com uma variável normal, esse tipo especial de variável é conhecido como **propriedade** (::property::). Comumente também são chamadas de variáveis de classe, variáveis membro, atributos ou ::field::s.

As propriedades de uma classe também podem ser inicializadas, caso necessário. Em cada novo produto, podemos explicitamente declarar o valor da propriedade $usado como false. Neste caso, nosso código ficaria assim:

* 1. class Produto
  2. {
  3. public $id;
  4. public $nome;
  5. public $preco;
  6. public $descricao;
  7. public $categoria;
  8. public $usado = false;
  9. }

Há um detalhe importante, o valor padrão precisa ser uma expressão constante. Não podemos, por exemplo, atribuir uma variável, uma outra propriedade ou mesmo uma chamada de função. O seguinte código é **inválido**:

* 1. class Produto
  2. {
  3. // inválido, não pode receber uma variável
  4. public $id = $algum\_atributo;
  5. // ok, isso é válido
  6. public $nome = 'Iphone 6 dobrável';
  7. // inválido, não pode receber outra propriedade
  8. public $preco = $nome;
  9. // inválido, tem que ser constante
  10. public $descricao = 'alguma '.'descricao';
  11. // inválido, não pode receber uma função
  12. public $categoria = retornaCategoriaPadrao();
  13. }

**Classes e objetos são a mesma coisa?**

É completamente natural confundir esses dois conceitos no início, mas é fundamental saber distinguir o que é classe e o que é objeto. E na verdade, veremos que isso é bem simples.

Vimos que uma classe é como um protótipo, um molde para criar um **objeto** de determinado tipo. Mas como isso funciona? A idéia é que nossa loja poderá ter zilhares de produtos, mas haverá apenas uma classe Produto. Um único molde. A partir dele diversos objetos serão criados. Uma analogia interessante é comparar uma classe com uma receita de bolo. A pergunta é certeira: você come uma receita de bolo? Não. Precisamos **instanciá-la**, criar um**objeto** bolo a partir dessa especificação (a classe) para utilizá-la. Podemos criar centenas de bolos a partir dessa classe (a receita, no caso), eles podem ser bem semelhantes, alguns até idênticos, mas são objetos diferentes.

Já vimos como criar uma classe, mas como utilizá-la?

**Criando e usando objetos**

Queremos a partir do molde Produto, construir (**instanciar**) um novo objeto desse tipo. Para fazer isso basta utilizar a palavra reservada new, seguida do nome da classe e **opcionalmente** os parênteses como a seguir:

* 1. $produto = new Produto();

Note que criamos uma variável $produto para que futuramente possamos nos referenciar a esse novo objeto que está sendo criado. Através dessa variável podemos acessar a atribuir valores a ele, como por exemplo:

* 1. $produto->nome = "Livro da Casa do Código";

Utilizamos o operador ->, conhecido como ::Object Operator::, para fazer essa atribuição. Da mesma forma podemos popular as propriedades preco, descricao, categoria e usado. Um exemplo seria:

* 1. $produto->nome = "Livro da Casa do Código";
  2. $produto->preco = 29.90;
  3. $produto->descricao = "livro de tecnologia";
  4. $produto->usado = true;
  5. // Trataremos da categoria mais a frente

É importante perceber que, mesmo que as propriedades tenham sido declaradas com o prefixo $, nós não usamos esse simbolo ao atribuir valores para elas. Sendo assim, o seguinte código é **inválido**:

* 1. <?php
  2. // Código inválido!!!
  3. $produto->$nome = "Livro da Casa do Código";

**Colocando o conhecimento em prática**

Agora que já temos uma base, vamos modificar nosso arquivo adiciona-produto.php para utilizar nossa classeProduto. No lugar de fazer:

* 1. $nome = $\_POST["nome"];
  2. $preco = $\_POST["preco"];
  3. $descricao = $\_POST["descricao"];
  4. $categoria\_id = $\_POST["categoria\_id"];
  5. $usado = $\_POST['usado'];

Vamos criar um novo objeto do tipo Produto, e atribuir esses valores a ele. O código deve ficar assim:

* 1. $produto = new Produto();
  2. $produto->nome = $\_POST["nome"];
  3. $produto->preco = $\_POST["preco"];
  4. $produto->descricao = $\_POST["descricao"];
  5. $produto->categoria\_id = $\_POST["categoria\_id"];
  6. $produto->usado = $\_POST['usado'];

Lembrando que para usar a classe Produto, precisaremos fazer o importe do arquivo produto.php. Para isso basta adicionar a linha:

* 1. <?php require\_once 'produto.php' ?>

Feito isso, em nosso if no lugar de fazer:

* 1. if (insereProduto($conexao, $nome, $preco, $descricao, $categoria\_id, $usado)) { ?>

Podemos passar apenas o $produto como argumento:

* 1. if (insereProduto($conexao, $produto)) { ?>

Muito mais simples e declarativo, não acha? Vamos agora modificar a função insereProduto, no arquivo banco-produto.php, para que passe a funcionar dessa forma. O código agora está assim:

* 1. function insereProduto($conexao, $nome, $preco, $descricao, $categoria\_id, $usado)
  2. {
  3. if(array\_key\_exists('usado', $\_POST)) {
  4. $usado = "true";
  5. } else {
  6. $usado = "false";
  7. }
  8. $query = "insert into produtos (nome, preco, descricao, categoria\_id, usado)
  9. values ('{$nome}', '{$preco}', '{$descricao}', '{$categoria\_id}', {$usado})";
  10. return mysqli\_query($conexao, $query);
  11. }

A primeira mudança será na assinatura dessa função, que deve ficar:

* 1. function insereProduto($conexao, $produto)
  2. {
  3. // código omitido
  4. }

E agora, em todo lugar que utilizavamos as variáveis diretamente, passaremos a usar a sintaxe do ::object operator:: (->) que acabamos de conhecer. O código deve ficar assim:

* 1. function insereProduto($conexao, $produto)
  2. {
  3. if(array\_key\_exists('usado', $\_POST)) {
  4. $produto->usado = "true";
  5. } else {
  6. $produto->usado = "false";
  7. }
  8. $query = "insert into produtos (nome, preco, descricao, categoria\_id,
  9. usado) values ('{$produto->nome}', '{$produto->preco}',
  10. '{$produto->descricao}', '{$produto->categoria\_id}', {$produto->usado})";
  11. return mysqli\_query($conexao, $query);
  12. }

Tudo pronto, rode o código para ver o resultado. Tudo continua funcionando.

É interessante desde já perceber que essa simples mudança proporciona uma manutenibilidade e legibilidade melhor em nosso código. Se precisarmos adicionar um novo campo no Produto, a assinatura da função não vai mudar. Nenhum lugar que chama a função insereProduto precisará ser tocado.

**PHP Orientado a Objetos**

Apesar do PHP não ser uma linguagem puramente orientada a objetos, ela possui suporte à programação orientada a objetos e cada vez mais a comunidade e bibliotecas tem utilizado essa abordagem. A orientação a objetos é um paradigma que busca tornar seu código mais organizado e reutilizável, dividindo responsabilidades de forma clara dentro do sistema.

Muitos imaginam que as linguagens orientadas a objetos são coisas novas e que só apareceram da década de 90 pra cá. Na verdade, a orientação a objetos é um paradigma antigo e que foi implementado inicialmente na linguagem ::Smalltalk::. Uma linguagem que surgiu na década de 70, apenas para estudos. Teve uma versão real da linguagem nos anos 80 e com os estudos e evolução da linguagem, outras foram seguindo o mesmo paradigma. C++, Java,C# e Ruby são algumas delas.

Mesmo com o grande incentivo da comunidade PHP para que ela se torne uma linguagem puramente orientada a objetos, muito legado ainda existe, o que dificulta a mudança de toda a linguagem e ferramentas. Neste capítulo, vimos algumas das muitas vantagens desse paradigma. Colocaremos tudo isso em prática em nossa aplicação.