Flexxo

Centro de Capacitação em TI

PHP Developer

Sumário

Fundamentos de Aplicações Web

Esta parte do curso pretende abordar o desenvolvimento de interfaces para Aplicações Web que acessamos por meio de navegadores, utilizando padrões atuais de desenvolvimento e conhecendo a suas características técnicas.

Discutiremos as implementações dessas tecnologias nos diferentes navegadores, a adoção de frameworks que facilitam e aceleram nosso trabalho.

HTML

A única linguagem que o navegador consegue interpretar para a exibição de conteúdo é o HTML.

Para iniciar a exploração do HTML, vamos imaginar o seguinte caso: o navegador realizou uma requisição e recebeu como corpo da resposta o seguinte conteúdo:

> *Olá Estudante,*
>
> *Seja bem vindo a esta página.*

Para conhecer o comportamento dos navegadores quanto ao conteúdo descrito antes, vamos reproduzir esse conteúdo em um arquivo de texto comum, que pode ser criado com qualquer editor de texto puro. Salve o arquivo como **index.html** e abra-o a partir do navegador à sua escolha.

Analisando o resultado exibido pelo navegador; apesar deste ser capaz de exibir texto puro em sua área principal, algumas regras devem ser seguidas caso desejemos que esse texto seja exibido com alguma formatação, para facilitar a leitura pelo usuário final.

Podemos concluir que o navegador por padrão:

- Pode não exibir caracteres acentuados corretamente;
- Não exibe quebras de linha.

Para que possamos exibir as informações com alguma formatação, é necessário que cada trecho de texto tenha uma marcação indicando qual é o significado dele. Essa marcação também influencia a maneira com que cada trecho do texto será exibido. A seguir é listado o texto com uma marcação correta:

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8">
 </head>
 <body>
  <h1>Olá Estudante,</h1>
  <h2>Seja bem vindo a esta página.</h2>
 </body>
</html>
```

Reproduza o código anterior em um novo arquivo de texto puro e salve-o como index-2.html e abra o arquivo no navegador.

Agora, o resultado é exibido de maneira muito mais agradável e legível. Para isso, tivemos que utilizar algumas marcações do HTML. Essas marcações são chamadas de **tags**, e elas basicamente dão significado ao texto contido entre sua abertura e fechamento.

Sintaxe do HTML

O HTML é um conjunto de tags responsáveis pela marcação do conteúdo de uma página no navegador. No código que vimos antes, as tags são os elementos a mais que escrevemos usando a sintaxe **<nomedatag>**.

Diversas tags são disponibilizadas pela linguagem HTML e cada uma possui uma funcionalidade específica.

No código de antes, vimos, por exemplo, o uso da tag **<h1>**. Ela representa o título principal da página.

```
<h1>Olá Estudante,</h1>
```

Note a sintaxe. Uma tag é definida com caracteres < e >, e seu nome (H1 no caso).

Muitas tags possuem conteúdo, como o texto do título ("Olá Estudante,"). Nesse caso, para determinar onde o conteúdo acaba, usamos uma tag de fechamento com a barra antes do nome: </h1>.

Algumas tags podem receber atributos dentro de sua definição. São parâmetros usando a sintaxe de nome=”valor”. Para definir uma imagem, por exemplo, usamos a tag <img> e, para indicar qual imagem carregar, usamos o atributo src:

```
<img src="./imagens/logo.jpg">
```

Repare que a tag img não possui conteúdo por si só. Nesses casos, não é necessário usar uma tag de fechamento como antes no h1.

ESTRUTURA DE UM DOCUMENTO HTML

Um documento HTML válido precisa seguir obrigatoriamente a estrutura composta pelas tags <html>, <head> e <body> e a instrução <!DOCTYPE>. Vejamos cada uma delas:

A tag <html>

Na estrutura do nosso documento, antes de tudo, inserimos uma tag <html>. Dentro dessa tag, é necessário declarar outras duas tags: <head> e <body>. Essas duas tags são "irmãs", pois estão no mesmo nível hierárquico em relação à sua tag "pai", que é <html>.

```
<html>
<head></head>
<body></body>
</html>
```

A tag <head>

A tag <head> contém informações sobre nosso documento que são de interesse somente do navegador, e não dos visitantes do nosso site. São informações que não serão exibidas na área do documento no navegador.

A especificação obriga a presença da tag de conteúdo <title> dentro do nosso <head>, permitindo especificar o título do nosso documento, que normalmente será exibido na barra de título da janela do navegador ou na aba do documento.

Outra configuração muito utilizada, principalmente em documentos cujo conteúdo é escrito em um idioma como o português, que tem caracteres como acentos e cedilha, é a configuração da codificação de caracteres, chamado de encoding ou charset.

Podemos configurar qual codificação queremos utilizar em nosso documento por meio da configuração de charset na tag <meta>. Um dos valores mais comuns usados hoje em dia é o UTF-8, também chamado de Unicode. Há outras possibilidades, como o latin1, muito usado antigamente.

O UTF-8 é a recomendação atual para encoding na Web por ser amplamente suportada em navegadores e editores de código, além de ser compatível com praticamente todos os idiomas do mundo. É o que usaremos no curso.

```
<html>
<head>
<title>Título Site</title>
<meta charset="utf-8">
</head>
<body>
</body>
</html>
```

A tag <body>

A tag <body> contém o corpo do nosso documento, que é exibido pelo navegador em sua janela. É necessário que o <body> tenha ao menos um elemento "filho", ou seja, uma ou mais tags HTML dentro dele.

```
<html>
<head>
<title>Título Site</title>
<meta charset="utf-8">
</head>
<body>
<h1>A Título Site</h1>
</body>
</html>
```

Nesse exemplo, usamos a tag <h1>, que indica um título.

A instrução DOCTYPE

O DOCTYPE não é uma tag HTML, mas uma instrução especial. Ela indica para o navegador qual versão do HTML deve ser utilizada para renderizar a página.

Utilizaremos <!DOCTYPE html>, que indica para o navegador a utilização da versão mais recente do HTML - a versão 5, atualmente.

Há muitos comandos complicados nessa parte de DOCTYPE que eram usados em versões anteriores do HTML e do XHTML. Hoje em dia, nada disso é mais importante. O recomendado é sempre usar a última versão do HTML, usando a declaração de DOCTYPE simples:

```
<!DOCTYPE html>
```

TAGS HTML

O HTML é composto de diversas tags, cada uma com sua função e significado. O HTML 5, então, adicionou muitas novas tags, que veremos ao longo do curso. Nesse momento, vamos focar em tags que representam títulos, parágrafo, ênfase, imagem, listas, hiperlinks e elementos estruturais.

Títulos

Quando queremos indicar que um texto é um título em nossa página, utilizamos as tags de heading em sua marcação:

```
<h1>Título Site.</h1>
<h2>Bem-vindo</h2>
```

As tags de heading são tags de conteúdo e vão de <h1> a <h6>, seguindo a ordem de importância, sendo <h1> o título principal, o mais importante, e <h6> o título de menor importância.

Utilizamos, por exemplo, a tag <h1> para o nome, título principal da página, e a tag <h2> como subtítulo ou como título de seções dentro do documento.

A ordem de importância, além de influenciar no tamanho padrão de exibição do texto, tem impacto nas ferramentas que processam HTML. As ferramentas de indexação de conteúdo para buscas, como o Google, Bing ou Yahoo! levam em consideração essa ordem e relevância. Os navegadores especiais para acessibilidade também interpretam o conteúdo dessas tags de maneira a diferenciar seu conteúdo e facilitar a navegação do usuário pelo documento.

Parágrafos

Quando exibimos qualquer texto em nossa página, é recomendado que ele seja sempre conteúdo de alguma tag filha da tag <body>. A marcação mais indicada para textos comuns é a tag de parágrafo:

```
<p>Texto exibido em um parágrafo</p>
```

Se você tiver vários parágrafos de texto, use várias dessas tags <p> para separá-los:

```
<p>Um parágrafo de texto.</p>
<p>Outro parágrafo de texto.</p>
```

Marcações de ênfase

Quando queremos dar uma ênfase diferente a um trecho de texto, podemos utilizar as marcações de ênfase. Podemos deixar um texto "mais forte" com a tag ou deixar o texto com uma "ênfase acentuada" com a tag . Também há a tag <small>, que diminui o tamanho do texto.

Por padrão, os navegadores renderizarão o texto dentro da tag em negrito e o texto dentro da tag em itálico. Existem ainda as tags <b> e <i>, que atingem o mesmo resultado visualmente, mas as tags e são mais indicadas por definirem nossa intenção de significado ao conteúdo, mais do que uma simples indicação visual. Vamos discutir melhor a questão do significado das tags mais adiante.

```
<p>Visite <strong>Título Site</strong>.</p>
```

Imagens

A tag <img> define uma imagem em uma página HTML e necessita de dois atributos preenchidos: src e alt. O primeiro aponta para o local da imagem e o segundo, um texto alternativo para a imagem caso essa não possa ser carregada ou visualizada.

O HTML 5 introduziu duas novas tags específicas para imagem: <figure> e <figcaption>. A tag <figure> define uma imagem com a conhecida tag <img>. Além disso, permite adicionar uma legenda para a imagem por meio da tag <figcaption>.

```
<figure>
<img src="img/produto1.png" alt="Foto do produto">
<figcaption>Fuzz Cardigan por R$ 129,90</figcaption>
</figure>
```

Listas

Não são raros os casos em que queremos exibir uma listagem em nossas páginas. O HTML tem algumas tags definidas para que possamos fazer isso de maneira correta. A lista mais comum é a lista não-ordenada.

```
<ul>
<li>Primeiro item da lista</li>
<li>
Segundo item da lista:
<ul>
<li>Primeiro item da lista aninhada</li>
<li>Segundo item da lista aninhada</li>
</ul>
</li>
<li>Terceiro item da lista</li>
</ul>
```

Note que, para cada item da lista não-ordenada, utilizamos uma marcação de item de lista <li>. No exemplo acima, utilizamos uma estrutura composta na qual o segundo item da lista contém uma nova lista. A mesma tag de item de lista <li> é utilizada quando demarcamos uma lista ordenada.

```
<ol>
<li>Primeiro item da lista</li>
<li>Segundo item da lista</li>
<li>Terceiro item da lista</li>
<li>Quarto item da lista</li>
<li>Quinto item da lista</li>
</ol>
```

As listas ordenadas também podem ter sua estrutura composta por outras listas ordenadas como no exemplo que temos para as listas não-ordenadas. Também é possível ter listas ordenadas aninhadas em um item de uma lista não-ordenada e vice-versa.

Existe um terceiro tipo de lista que devemos utilizar para demarcar um glossário, quando listamos termos e seus significados. Essa lista é a lista de definição.

```
<dl>
<dt>HTML</dt>
<dd>
HTML é a linguagem de marcação de textos utilizada
para exibir textos como páginas na Internet.
</dd>
<dt>Navegador</dt>
<dd>
```

```
Navegador é o software que requisita um documento HTML
através do protocolo HTTP e exibe seu conteúdo em uma
janela.
</dd>
</dl>
```

Hiperlinks

Quando precisamos indicar que um trecho de texto se refere a um outro conteúdo, seja ele no mesmo documento ou em outro endereço, utilizamos a tag de âncora <a>.

Existem dois diferentes usos para as âncoras. Um deles é a definição de links:

```
<p>
Visite o site da <a href="http://www.google.com.br">Gogle</a>.
</p>
```

Note que a âncora está demarcando apenas a palavra "Google" de todo o conteúdo do parágrafo exemplificado. Isso significa que, ao clicarmos com o cursor do mouse na palavra "Google", o navegador redirecionará o usuário para o site da Google, indicado no atributo href.

Outro uso para a tag de âncora é a demarcação de destinos para links dentro do próprio documento, o que chamamos de bookmark.

```
<p>Mais informações <a href="#info">aqui</a>.</p>
<p>Conteúdo da página...</p>
<h2 id="info">Mais informações sobre o assunto:</h2>
<p>Informações...</p>
```

De acordo com o exemplo acima, ao clicarmos sobre a palavra "aqui", demarcada com um link, o usuário será levado à porção da página onde o bookmark "info" é visível. Bookmark é o elemento que tem o atributo id. É possível, com o uso de um link, levar o usuário a um bookmark presente em outra página.

```
<a href="http://www.dominio.com.br/#contato">
Entre em contato
</a>
```

O exemplo acima fará com que o usuário que clicar no link seja levado à porção da página indicada no endereço, especificamente no ponto onde o bookmark "contato" seja visível.

Elementos Estruturais

Já vimos muitas tags para casos específicos: títulos com h1, parágrafos com p, imagens com img, listas com ul etc. E ainda vamos ver várias outras.

Mas é claro que não existe uma tag diferente para cada coisa do universo. O conjunto de tags do HTML é bem vasto, mas é também limitado.

Invariavelmente você vai cair algum dia num cenário onde não consegue achar a tag certa para aquele conteúdo. Nesse caso, pode usar as tags e que funcionam como coringas. São tags sem nenhum significado especial, mas que podem servir para agrupar um certo conteúdo, tanto um bloco da página quanto um pedaço de texto.

E, como vamos ver a seguir, vamos poder estilizar esses divs e spans com CSS customizado. Por padrão, eles não têm estilo algum.

CSS

Quando escrevemos o HTML, marcamos o conteúdo da página com tags que melhor representam o significado daquele conteúdo. Aí quando abrimos a página no navegador é possível perceber que o navegador mostra as informações com estilos diferentes.

Um h1, por exemplo, fica em negrito numa fonte maior. Parágrafos de texto são espaçados entre si, e assim por diante. Isso quer dizer que o navegador tem um estilo padrão para as tags que usamos. Mas, claro, para fazer sites bonitões vamos querer customizar o design dos elementos da página.

Antigamente, isso era feito no próprio HTML. Se quisesse um título em vermelho, era só fazer:

```
<h1><font color="red">Título Site</font></h1>
```

Além da tag font, várias outras tags de estilo existiam. Mas isso é passado. Em seu lugar, surgiu o CSS, que é uma outra linguagem, separada do HTML, com objetivo único de cuidar da estilização da página. A vantagem é que o CSS é bem mais robusto que o HTML para estilização, como veremos. Mas, principalmente, porque escrever formatação visual misturado com conteúdo de texto no HTML se mostrou algo bem impraticável. O CSS resolve isso separando as coisas; regras de estilo não aparecem mais no HTML, apenas no CSS.

SINTAXE E INCLUSÃO DE CSS

A sintaxe do CSS tem estrutura simples: é uma declaração de propriedades e valores separados por um sinal de dois pontos (:), e cada propriedade é separada por um sinal de ponto e vírgula (;) da seguinte maneira:

```
background-color: yellow;
color: blue;
```

O elemento que receber essas propriedades será exibido com o texto na cor azul e com o fundo amarelo. Essas propriedades podem ser declaradas de três maneiras diferentes.

Atributo style

A primeira delas é como um atributo style no próprio elemento:

```
<p style="color: blue; background-color: yellow;">
O conteúdo desta tag será exibido em azul com fundo amarelo no navegador!
</p>
```

Mas tínhamos acabado de discutir que uma das grandes vantagens do CSS era manter as regras de estilo fora do HTML. Usando esse atributo style não parece que fizemos isso. Justamente por isso não se recomenda esse tipo de uso na prática, mas sim os que veremos a seguir.

A tag style

A outra maneira de se utilizar o CSS é declarando suas propriedades dentro de uma tag <style>.

Como estamos declarando as propriedades visuais de um elemento em outro lugar do nosso documento, precisamos indicar de alguma maneira a qual elemento nos referimos. Fazemos isso utilizando um seletor CSS. É basicamente uma forma de buscar certos elementos dentro da página que receberão as regras visuais que queremos.

No exemplo a seguir, usaremos o seletor que pega todas as tags p e altera sua cor e background:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Título Site</title>
<style>
p {
background-color: yellow;
color: blue;
}
</style>
</head>
<body>
<p>
O conteúdo desta tag será exibido em azul com fundo amarelo!
</p>
<p>
<strong>Também</strong> será exibido em azul com fundo amarelo!
</p>
</body>
</html>
```

O código anterior da tag <style> indica que estamos alterando a cor e o fundo de todos os elementos com tag p. Dizemos que selecionamos esses elementos pelo nome de sua tag, e aplicamos certas propriedades CSS apenas neles.

Arquivo externo

A terceira maneira de declararmos os estilos do nosso documento é com um arquivo externo, geralmente com a extensão .css. Para que seja possível declarar nosso CSS em um arquivo à parte, precisamos indicar em nosso documento HTML uma ligação entre ele e a folha de estilo.

Além da melhor organização do projeto, a folha de estilo externa traz ainda as vantagens de manter nosso HTML mais limpo e do reaproveitamento de uma mesma folha de estilos para diversos documentos.

A indicação de uso de uma folha de estilos externa deve ser feita dentro da tag <head> do nosso documento HTML:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title> Título Site</title>
```

```
<link rel="stylesheet" href="estilos.css">
</head>
<body>
<p>
O conteúdo desta tag será exibido em azul com fundo amarelo!
</p>
<p>
<strong>Também</strong> será exibido em azul com fundo amarelo!
</p>
</body>
</html>
```

E dentro do arquivo estilos.css colocamos apenas o conteúdo do CSS:

```
p {
color: blue;
background-color: yellow;
}
```

PROPRIEDADES TIPOGRÁFICAS E FONTES

Da mesma maneira que alteramos cores, podemos alterar o texto. Podemos definir fontes com o uso da propriedade font-family.

A propriedade font-family pode receber seu valor com ou sem aspas. No primeiro caso, passaremos o nome do arquivo de fonte a ser utilizado, no último, passaremos a família da fonte.

Por padrão, os navegadores mais conhecidos exibem texto em um tipo que conhecemos como "serif". As fontes mais conhecidas (e comumente utilizadas como padrão) são "Times" e "Times New Roman", dependendo do sistema operacional. Elas são chamadas de fontes serifadas pelos pequenos ornamentos em suas terminações.

Podemos alterar a família de fontes que queremos utilizar em nosso documento para a família "sans-serif" (sem serifas), que contém, por exemplo, as fontes "Arial" e "Helvetica". Podemos também declarar que queremos utilizar uma família de fontes "monospace" como, por exemplo, a fonte "Courier".

```
h1 {
font-family: serif;
}
h2 {
font-family: sans-serif;
}
p {
```

```
font-family: monospace;
}
```

É possível, e muito comum, declararmos o nome de algumas fontes que gostaríamos de verificar se existem no computador, permitindo que tenhamos um controle melhor da forma como nosso texto será exibido. Normalmente, declaramos as fontes mais comuns, e existe um grupo de fontes que são consideradas "seguras" por serem bem populares.

```
body {
font-family: "Arial", "Helvetica", sans-serif;
}
```

Nesse caso, o navegador verificará se a fonte "Arial" está disponível e a utilizará para renderizar os textos de todos os elementos do nosso documento que, por cascata, herdarão essa propriedade do elemento body.

Caso a fonte "Arial" não esteja disponível, o navegador verificará a disponibilidade da próxima fonte declarada, no nosso exemplo a "Helvetica".

Caso o navegador não encontre também essa fonte, ele solicita qualquer fonte que pertença à família "sans-serif", declarada logo a seguir, e a utiliza para exibir o texto, não importa qual seja ela.

Temos outras propriedades para manipular a fonte, como a propriedade fontstyle, que define o estilo da fonte que pode ser: normal (normal na vertical), italic (inclinada) e oblique (oblíqua).

ALINHAMENTO E DECORAÇÃO DE TEXTO

Já vimos uma série de propriedades e subpropriedades que determinam o estilo do tipo (fonte). Vamos conhecer algumas maneiras de alterarmos as disposições dos textos.

Uma das propriedades mais simples, porém, muito utilizada, é a que diz respeito ao alinhamento de texto: a propriedade text-align.

```
p {
text-align: right;
}
```

O exemplo anterior determina que todos os parágrafos da nossa página tenham o texto alinhado para a direita. Também é possível determinar que um elemento tenha seu conteúdo alinhado ao centro ao definirmos o valor center para a propriedade text-align, ou

então definir que o texto deve ocupar toda a largura do elemento aumentando o espaçamento entre as palavras com o valor justify. O padrão é que o texto seja alinhado à esquerda, com o valor left, porém é importante lembrar que essa propriedade propaga-se em cascata.

É possível configurar também uma série de espaçamentos de texto com o CSS:

```
p {
line-height: 3px; /* tamanho da altura de cada linha */
letter-spacing: 3px; /* tamanho do espaço entre cada letra */
word-spacing: 5px; /* tamanho do espaço entre cada palavra */
text-indent: 30px; /* tamanho da margem da primeira linha do texto */
}
```

IMAGEM DE FUNDO

A propriedade background-image permite indicar um arquivo de imagem para ser exibido ao fundo do elemento. Por exemplo:

```
h1 {
background-image: url(sobre-background.jpg);
}
```

Com essa declaração, o navegador vai requisitar o arquivo sobre-background.jpg, que deve estar na mesma pasta do arquivo CSS onde consta essa declaração.

BORDAS

As propriedades do CSS para definirmos as bordas de um elemento nos apresentam uma série de opções. Podemos, para cada borda de um elemento, determinar sua cor, seu estilo de exibição e sua largura. Por exemplo:

```
body {
border-color: red;
border-style: solid;
border-width: 1px;
}
```

Para que o efeito da cor sobre a borda surta efeito, é necessário que a propriedade border-style tenha qualquer valor diferente do padrão none.

CORES NA WEB

Propriedades como background-color, color, border-color, entre outras aceitam uma cor como valor. Existem várias maneiras de definir cores quando utilizamos o CSS.

A primeira, mais simples e ingênua, é usando o nome da cor:

```
h1 {
color: red;
}
h2 {
background: yellow;
}
```

O difícil é acertar a exata variação de cor que queremos no design. Por isso, é bem incomum usarmos cores com seus nomes. O mais comum é definir a cor com base em sua composição RGB.

RGB é um sistema de cor bastante comum aos designers. Ele permite especificar até 16 milhões de cores como uma combinação de três cores base: Vermelho (Red), Verde (Green), Azul (Blue).

Podemos escolher a intensidade de cada um desses três canais básicos, numa escala de 0 a 255.

Um amarelo forte, por exemplo, tem 255 de Red, 255 de Green e 0 de Blue (255, 255, 0). Se quiser um laranja, basta diminuir um pouco o verde (255, 200, 0). E assim por diante.

No CSS, podemos escrever as cores tendo como base sua composição RGB. Aliás, no CSS há até uma sintaxe bem simples para isso:

```
h3 {
color: rgb(255, 200, 0);
}
```

Essa sintaxe funciona nos browsers mais modernos, mas não é a mais comum na prática, por questões de compatibilidade. O mais comum é a notação hexadecimal, que acabamos usando no exercício anterior ao escrever #F2EDED.

Essa sintaxe tem suporte universal nos navegadores e é mais curta de escrever, apesar de ser mais enigmática.

```
h3 {
background: #F2EDED;
}
```

No fundo, porém, é a mesma coisa. Na notação hexadecimal (que começa com #), temos

6 caracteres. Os primeiros 2 indicam o canal Red, os dois seguintes, o Green, e os dois últimos, Blue. Ou seja, RGB. E usamos a matemática para escrever menos, trocando a base numérica de decimal para hexadecimal.

Na base hexadecimal, os algarismos vão de zero a quinze (ao invés do zero a nove da base decimal comum). Para representar os algarismos de dez a quinze, usamos letras de A à F. Nessa sintaxe, portanto, podemos utilizar números de 0-9 e letras de A-F.

Há uma conta por trás dessas conversões, mas seu editor de imagens deve ser capaz de fornecer ambos os valores para você sem problemas. Um valor 255 viras FF na notação hexadecimal. A cor #F2EDED, por exemplo, é equivalente a rgb(242, 237, 237), um cinza claro.

Vale aqui uma dica quanto ao uso de cores hexadecimais, toda vez que os caracteres presentes na composição da base se repetirem, estes podem ser simplificados. Então um número em hexadecimal 3366FF, pode ser simplificado para 36F.

ESPAÇAMENTO E MARGEM

Utilizamos a propriedade padding para espaçamento e margin para margem. Vejamos cada uma e como elas diferem entre si.

Padding

A propriedade padding é utilizada para definir uma margem interna em alguns elementos (por margem interna queremos dizer a distância entre o limite do elemento, sua borda, e seu respectivo conteúdo) e tem as subpropriedades listadas a seguir:

- padding-top
- padding-right
- padding-bottom
- padding-left

Essas propriedades aplicam uma distância entre o limite do elemento e seu conteúdo acima, à direita, abaixo e à esquerda respectivamente. Essa ordem é importante para entendermos como funciona a shorthand property do padding.

Podemos definir todos os valores para as subpropriedades do padding em uma única propriedade, chamada exatamente de padding, e seu comportamento é descrito nos exemplos a seguir:

Se passado somente um valor para a propriedade padding, esse mesmo valor é aplicado em todas as direções.

```
p {
padding: 10px;
}
```

Se passados dois valores, o primeiro será aplicado acima e abaixo (equivalente a passar o mesmo valor para padding-top e padding-bottom) e o segundo será aplicado à direita e à esquerda (equivalente ao mesmo valor para padding-right e padding-left).

```
p {
padding: 10px 15px;
}
```

Se passados três valores, o primeiro será aplicado acima (equivalente a padding-top), o segundo será aplicado à direita e à esquerda (equivalente a passar o mesmo valor para padding-right e padding-left) e o terceiro valor será aplicado abaixo do elemento (equivalente a padding-bottom).

```
p {
padding: 10px 20px 15px;
}
```

Se passados quatro valores, serão aplicados respectivamente a padding-top, padding-right, padding-bottom e padding-left. Para facilitar a memorização dessa ordem, basta lembrar que os valores são aplicados em sentido horário.

```
p {
padding: 10px 20px 15px 5px;
}
```

Margin

A propriedade margin é bem parecida com a propriedade padding, exceto que ela adiciona espaço após o limite do elemento, ou seja, é um espaçamento além do elemento em si. Além das subpropriedades listadas a seguir, há a shorthand property margin que se comporta da mesma maneira que a shorthand property do padding vista no tópico anterior.

- margin-top
- margin-right
- margin-bottom
- margin-left

Há ainda uma maneira de permitir que o navegador defina qual será a dimensão da propriedade padding ou margin conforme o espaço disponível na tela: definimos o valor auto para a margem ou o espaçamento que quisermos.

No exemplo a seguir, definimos que um elemento não tem nenhuma margem acima ou abaixo de seu conteúdo e que o navegador define uma margem igual para ambos os lados de acordo com o espaço disponível:

```
p {
margin: 0 auto;
}
```

Dimensões

É possível determinar as dimensões de um elemento, por exemplo:

```
p {
background-color: red;
height: 300px;
width: 300px;
}
```

Todos os parágrafos do nosso HTML ocuparão 300 pixels de largura e de altura, com cor de fundo vermelha.

Seletores de ID e Hierárquico

Já vimos como selecionar elementos no CSS usando simplesmente o nome da tag:

```
p {
color: red;
}
```

Apesar de simples, é uma maneira muito limitada de selecionar. Às vezes não queremos pegar todos os parágrafos da página, mas apenas algum determinado. Existem, portanto, maneiras avançadas de selecionarmos um ou mais elementos do HTML usando os seletores CSS. Vamos ver seletores CSS quase que ao longo do curso todo, inclusive alguns bem avançados e modernos do CSS3. Por enquanto, vamos ver mais dois básicos além do seletor por nome de tag.

Seletor de ID

É possível aplicar propriedades visuais a um elemento selecionado pelo valor de seu atributo id. Para isso, o seletor deve iniciar com o caractere "#" seguido do valor correspondente.

```
#cabecalho {
color: white;
text-align: center;
}
```

O seletor acima fará com que o elemento do nosso HTML que tem o atributo id com valor "cabeçalho" tenha seu texto renderizado na cor branca e centralizado.

Note que não há nenhuma indicação para qual tag a propriedade será aplicada.

Pode ser tanto uma quanto um <p>, até mesmo tags sem conteúdo como uma <img>, desde que essa tenha o atributo id com o valor "cabeçalho".

Como o atributo id deve ter valor único no documento, o seletor deve aplicar suas propriedades declaradas somente àquele único elemento e, por cascata, a todos os seus elementos filhos.

Seletor hierárquico

Podemos ainda utilizar um seletor hierárquico que permite aplicar estilos aos elementos filhos de um elemento pai:

```
#rodape img {
margin-right: 35px;
vertical-align: middle;
width: 94px;
}
```

No exemplo anterior, o elemento pai rodapé é selecionado pelo seu id. O estilo será aplicado apenas nos elementos img filhos do elemento com id=rodapé.

JavaScript

O JavaScript, como o próprio nome sugere, é uma linguagem de scripting. Uma linguagem de scripting é comumente definida como uma linguagem de programação que permite ao programador controlar uma ou mais aplicações de terceiros. No caso do JavaScript, podemos controlar alguns comportamentos dos navegadores através de trechos de código que são enviados na página HTML.

Outra característica comum nas linguagens de scripting é que normalmente elas são linguagens interpretadas, ou seja, não dependem de compilação para serem executadas. Essa característica é presente no JavaScript: o código é interpretado e executado conforme é lido pelo navegador, linha a linha, assim como o HTML.

O JavaScript também possui grande tolerância a erros, uma vez que conversões automáticas são realizadas durante operações. Como será visto no decorrer das explicações, nem sempre essas conversões resultam em algo esperado, o que pode ser fonte de muitos bugs, caso não conheçamos bem esse mecanismo.

O script do programador é enviado com o HTML para o navegador, mas como o navegador saberá diferenciar o script de um código html? Para que essa diferenciação seja possível, é necessário envolver o script dentro da tag <script>.

A TAG SCRIPT

Para rodar JavaScript em uma página Web, precisamos ter em mente que a execução do código é instantânea. Para inserirmos um código JavaScript em uma página, é necessário utilizar a tag <script>:

```
<script>
alert("Olá, Mundo!");
</script>
```

O exemplo acima é um "hello world" em JavaScript utilizando uma função do navegador, a função alert.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Aula de JS</title>
<script>
alert("Olá, Mundo!");
```

```
</script>
</head>
<body>
<h1>JavaScript</h1>
<h2>Linguagem de programação</h2>
</body>
</html>
```

Repare que, ao ser executado, o script trava o processamento da página. Imagine um script que demore um pouco mais para ser executado ou que exija alguma interação do usuário como uma confirmação. Não seria interessante carregar a página toda primeiro antes de sua execução por uma questão de performance e experiência para o usuário? Para fazer isso, basta removermos o script do head, colocando-o no final do body:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Aula de JS</title>
</head>
<body>
<h1>JavaScript</h1>
<h2>Linguagem de Programação</h2>
<script>
alert("Olá, Mundo!");
</script>
</body>
</html>
```

Devemos sempre colocar o script antes de fecharmos a tag </body>? Na maioria esmagadora das vezes sim.

JavaScript em arquivo externo

Imagine ter que escrever o script toda vez que ele for necessário. Se ele for utilizado em outra página? Por isso é possível importar scripts dentro da página utilizando também a tag <script>:

No arquivo HTML

```
<script src="js/hello.js"></script>
```

Arquivo externo js/hello.js

```
alert("Olá, Mundo!");
```

CONSOLE DO NAVEGADOR

Alguns navegadores dão suporte à entrada de comandos pelo que chamamos de console. O console nos permite testar códigos diretamente no navegador sem termos que colocar uma tag <script> na página.

Experimente executar um alert no console e veja o resultado:

```
alert("interagindo com o console!");
```

SINTAXE BÁSICA

OPERADORES

Podemos somar, subtrair, multiplicar e dividir como em qualquer linguagem:

Teste algumas contas digitando diretamente no console:

```
> 12 + 13
25
```

```
> 14 * 3
42
```

```
> 10 - 4
6
```

```
> 25 / 5
5
```

```
> 23 % 2
1
```

Variáveis

Para armazenarmos um valor para uso posterior, podemos criar uma variável:

```
var curso = "Curso de Javascript";
alert(curso);
```

No exemplo acima, guardamos o valor “Curso de Javascript” na variável curso. A partir desse ponto, é possível utilizar a variável para obter o valor que guardamos nela ou mudar o seu valor, como veremos na próxima seção.

Também podemos alterar o valor de uma variável usando essas operações com uma sintaxe bem compacta:

```
var idade = 10;
idade += 10; // idade vale 20
idade -= 5; // idade vale 15
idade /= 3; // idade vale 5
idade *= 10; // idade vale 50
```

Mensagens secretas no console

É comum querermos dar uma olhada no valor de alguma variável ou resultado de alguma operação durante a execução do código. Nesses casos, poderíamos usar um alert, porém, se esse conteúdo deveria somente ser mostrado para o desenvolvedor, o console do navegador pode ser utilizado no lugar do alert para imprimir essa mensagem:

```
var mensagem = "Olá mundo";
console.log(mensagem);
```

TIPOS DE DADOS

Os principais tipos de dados em JavaScript são: string, number e boolean. Há também um tipo complexo chamado objeto, que veremos mais para frente.

String

Uma string em JavaScript é utilizada para armazenar trechos de texto:

```
var empresa = "BRASIL";
console.log(empresa);
```

Uma variável que armazena uma string faz muito mais que isso! Ela permite, por exemplo, consultar o seu tamanho e realizar transformações em seu valor.

```
var curso = "tecnologia";
curso.length; // tamanho da string
curso.replace("tecno","socio"); // retorna sociologia
```

A partir da variável curso, usamos o operador ponto seguido da ação replace. Essa ação é o que chamamos de função.

Função é um trecho de código guardado que pode receber um ou mais parâmetros separados por vírgula através do ( ).

Imutabilidade

String é imutável. Logo, no exemplo abaixo, se a variável empresa for impressa após a chamada da função replace, o valor continuará sendo "Nome da Empresa ". Para obter uma String modificada, é necessário receber o retorno de cada função que manipula a String, pois uma nova String modificada é retornada:

```
var empresa = "Nome da Empresa";
// substitui a parte "Empresa" por "Sociedade"
empresa.replace("Empresa","Sociedade");
console.log(empresa); // imprime Nome da Empresa, não mudou
empresa = empresa.replace("Empresa","Sociedade");
console.log(empresa); // imprime Nome da Sociedade, mudou!
```

Number

O JavaScript tem um tipo para tratar de todos os números: o tipo Number. Ele realiza operações aritméticas com maior facilidade, pois todas as conversões necessárias são realizadas automaticamente pelo interpretador.

```
var vinte = 20;
var pi = 3.14159;
```

Boolean

O tipo boolean não traz funções especiais e guarda apenas os valores true e false:

```
var b1 = true;
console.log(b1);
b1 = false;
```

```
console.log(b1);
```

Conversões

O JavaScript possui funções de conversão de string para number:

```
var textoInteiro = "10";
var inteiro = parseInt(textoInteiro);
var textoFloat = "10.22";
var float = parseFloat(textoFloat);
```

Number, assim como String, também é imutável. O exemplo abaixo altera o número de casas decimais com a função toFixed. Esta função retorna uma string mas, para ela funcionar corretamente, seu retorno precisa ser capturado:

```
var milNumber = 1000;
var milString = milNumber.toFixed(2); // recebe o retorno da função
console.log(milString); // imprime a string "1000.00"
```

Concatenações

É possível concatenar (juntar) tipos diferentes e o JavaScript se encarregará de realizar a conversão entre os tipos, podendo resultar em algo não esperado.

String com String

```
var s1 = "Brasil";
var s2 = "2016";
console.log(s1 + s2); // imprime Brasil2016
```

String com outro tipo de dados

Como vimos, o JavaScript tentará ajudar realizando conversões quando tipos diferentes forem envolvidos numa operação, mas é necessário estarmos atentos na maneira como ele as realiza:

```
var num1 = 2;
var num2 = 3;
var nome = "Brasil"
// Exemplo 1:
```

```
console.log(num1 + nome + num2); // imprime 2Brasil3
// Exemplo 2:
console.log(num1 + num2 + nome); // imprime 5Brasil
// Exemplo 3:
console.log(nome + num1 + num2); // imprime Brasil23
// Exemplo 4:
console.log(nome + (num1 + num2)); // imprime Brasil5
// Exemplo 5:
console.log(nome + num1 * num2); // imprime Brasil6.
// A multiplicação tem precedência
```

NaN

Veja o código abaixo:

```
console.log(10-"brasil”)
```

O resultado é NaN (not a number). Isto significa que todas operações matemáticas, exceto subtração, que serão vistas mais à frente, só podem ser feitas com números. O valor NaN ainda possui uma peculiaridade, definida em sua especificação:

```
var resultado = 10-"brasil"; // retorna NaN
resultado == NaN; // false
NaN == NaN; // false
```

Não é possível comparar uma variável com NaN, nem mesmo NaN com NaN! Para saber se uma variável é NaN, deve ser usada a função isNaN:

```
var resultado = 10-"brasil";
isNaN(resultado); // true
```

COMPARAÇÕES

Veja o seguinte exemplo abaixo:

```
var num2 = 10;
console.log(num1 == num2); // imprime true
console.log(num1 != num2); // imprime false
```

Ambas retornam um valor booleano. Podemos ainda usar <, <=, >,>=.

O que acontecerá com a comparação abaixo?

```
console.log(1 == "1"); // retorna true
```

O JavaScript também realiza conversões automáticas em comparações. Quando a comparação envolve uma String e um número, ele converte a String em número. A mesma coisa acontece quando um número é comparado com um boolean, ou seja, o booleano é convertido para número.

A mesma coisa acontece quando testamos e um número é maior que outro:

```
console.log(2 > "1"); // retorna true
```

Nem sempre esse é o comportamento desejado, por isso existe o operador === que verifica se um objeto é exatamente igual ao outro, isto é, se o seu valor e o tipo são os mesmos.

Refazendo os exemplos anteriores:

```
console.log(1 === "1"); // retorna false
```

Utiliza-se!== quando queremos o resultado da comparação exatamente diferente.

BLOCOS CONDICIONAIS

O bloco condicional mais comum é o if que recebe um valor booleano como parâmetro e possui a seguinte estrutura:

```
var idade = 18;
var temCarteira = true;
if (idade >= 18) {
alert("Pode dirigir");
} else {
alert("Não pode");
}
```

Ainda é possível utilizar os operadores && (E) e o || (OU):

```
var temCarteira = true;
if (idade >= 18 && temCarteira) {
alert("Pode dirigir");
} else {
alert("Não pode");
}
```

```
var temCarteiraDeTrabalho = false;
if (idade >= 18 || temCarteiraDeTrabalho) {
alert("Pode trabalhar");
```

```
} else {
alert("Não pode");
}
```

FUNÇÕES

A execução do JavaScript, quando interpretado pelo navegador, é imediata. O JavaScript é executado assim que uma tag <script> é encontrada em nosso documento.

No entanto, muitas vezes desejamos definir um comportamento para ser executado em outro momento, inclusive para que possa ser reaproveitado depois.

Para isso, existem as funções, com a seguinte estrutura básica:

```
// estrutura de uma função
function nomeDaFuncao(parametro) {
// corpo da função
};
```

A função criada desta forma é chamada de function declaration.

Criando uma função seguindo a estrutura acima:

```
function exibeMensagem() {
alert("Atenção");
}
// chamando a função declarada duas vezes!
exibeMensagem();
exibeMensagem();
```

No exemplo acima, a mensagem está fixa, mas pode ser interessante personalizar esta mensagem. É o que veremos a seguir.

Repare também que não foi necessário ponto e vírgula no final da declaração de função, apesar de nada impedir o programador de colocá-lo, caso queira.

Função com parâmetros

Uma função pode receber um ou mais parâmetros. Nosso primeiro exemplo exibe uma mensagem com o texto passado como parâmetro:

```
function exibeMensagem(mensagem) {
alert(mensagem);
}
exibeMensagem("JavaScript"); // exibe JavaScript
```

Geralmente as funções fazem mais do que exibir mensagens. A função abaixo exibe a soma de dois números passados como parâmetro:

```
function somaDoisNumeros(numero1, numero2){
alert(numero1 + numero2);
}
somaDoisNumeros(10, 20); // exibe 30
somaDoisNumeros(100, 20); // exibe 120
somaDoisNumeros(45, 35); // exibe 80
```

Função com retorno

No exemplo anterior, a função somaDoisNumeros exibe o resultado da soma, mas, muitas vezes, precisamos do resultado porque alguma operação do nosso código depende dele.

No exemplo abaixo, precisamos subtrair da variável numero o resultado da função somaDoisNumeros:

```
function somaDoisNumeros(numero1, numero2){
alert(numero1 + numero2);
}
var numero = 80;
// a função só exibe, não retorna o valor! Como subtrair 80 do valor calculado?
somaDoisNumeros(10,20); // exibe 30
```

Para que seja possível realizar a subtração, a função somaDoisNumeros precisa retornar o valor da soma, o que é possível através do comando return:

```
function somaDoisNumeros(numero1, numero2){
return numero1 + numero2;
}
var numero = 80;
var resultado = somaDoisNumeros(10,20); // armazena 30 na variável resultado
alert(numero - resultado); // exibe 50
```

Function Expression

Podemos declarar funções como conteúdo de variáveis, numa forma chamada function expression:

```
var somaDoisNumeros = function(numero1, numero2) {
return numero1 + numero2;
};
somaDoisNumeros(10,20);
```

Repare que a variável recebe como parâmetro um bloco de função sem nome, isto é, uma função anônima. Outra peculiaridade é que a declaração termina com ponto e vírgula. Como a função é anônima, a única maneira de invocá-la é através da variável que a guarda.

Function Expression Vs Function Declaration

As duas formas possuem o mesmo resultado, a diferença está em seu carregamento pelo navegador.

```
somaDoisNumeros(10, 20);
function somaDoisNumeros(num1, num2) {
return num1 + num2;
}
```

O código acima funciona, mesmo chamando a função antes de sua declaração. Já o código a seguir dá um erro:

```
somaDoisNumeros(10, 20); // erro, função não foi declarada aida
var somaDoisNumeros = function(num1, num2) {
return num1 + num2;
};
```

Isso ocorre devido a um processo chamado hoisting (levantamento, içamento). Function declarations são 'içadas' até o topo do script pelo JavaScript. As mesmas coisas são feitas com variáveis, mas a diferença é que apenas as declarações são 'içadas' até o topo do script, isto é, seu conteúdo continua como 'undefined'. É como se o interpretador fizesse isso:

```
var somaDoisNumeros;
somaDoisNumeros();
somaDoisNumeros = function(num1, num2) {
return num1 + num2;
};
```

INTERATIVIDADE

O uso do JavaScript como a principal linguagem de programação da Web só é possível por conta da integração que o navegador oferece para o programador, a maneira com que conseguimos encontrar um elemento da página e alterar alguma característica dele pelo código JavaScript:

```
var titulo = document.querySelector("#titulo");
titulo.textContent = "Agora o texto do elemento mudou!";
```

No exemplo anterior, nós selecionamos um elemento do documento e alteramos sua propriedade textContent. Existem diversas maneiras de selecionarmos elementos de um documento e de alterarmos suas propriedades.

Inclusive é possível selecionar elementos de maneira similar ao CSS, através de seletores CSS:

```
var painelNovidades = document.querySelector("section#main .painel#novidades");
painelNovidades.style.color = "red"
```

Apesar de ser interessante a possibilidade de alterar o documento todo por meio do JavaScript, existe um problema com a característica de execução imediata do código. O mais comum é que as alterações sejam feitas quando o usuário executa alguma ação, como por exemplo, clicar em um elemento.

Para suprir essa necessidade, é necessário utilizar de dois recursos do JavaScript no navegador. O primeiro é a criação de funções:

```
function mostraAlerta() {
alert("Funciona!");
}
```

Ao criarmos uma função, a execução do código simplesmente guarda o que estiver dentro da função, e esse código guardado só será executado quando chamarmos a função. Para exemplificar a necessidade citada acima, vamos utilizar o segundo recurso, os eventos:

```
// obtendo um elemento através de um seletor de ID
var titulo = document.querySelector("#titulo");
titulo.onclick = mostraAlerta;
```

Note que primeiramente é necessário selecionar um elemento do documento e depois definir o onclick desse elemento como sendo a função.

De acordo com os dois códigos acima, primeiramente guardamos um comportamento em uma função. Ao contrário do código fora de uma função, o comportamento não é executado imediatamente, e sim guardado para quando quisermos chamá-lo. Depois informamos que um elemento deve executar aquele comportamento em determinado momento, no caso em um evento "click".

Também é possível chamar uma função em um momento sem a necessidade de um evento, é só utilizar o nome da função abrindo e fechando parênteses, indicando que estamos executando a função que foi definida anteriormente:

```
function mostraAlerta() {
alert("Funciona!");
}
// Chamando a função:
mostraAlerta();
```

Também é possível determinar, por meio de seus argumentos, que a função vai ter algum valor variável que vamos definir quando quisermos executá-la:

```
function mostraAlerta(texto) {
// Dentro da função "texto" conterá o valor passado na execução.
alert(texto);
}
// Ao chamar a função é necessário definir o valor do "texto"
mostraAlerta("Funciona com argumento!");
```

Existem diversos eventos que podem ser utilizados para que a interação do usuário com a página seja o ponto de disparo de funções que alteram os elementos da própria página:

- onclick: clica com o mouse
- ondblclick: clica duas vezes com o mouse
- onmousemove: mexe o mouse
- onmousedown: aperta o botão do mouse
- onmouseup: solta o botão do mouse (útil com os dois acima para gerenciar drag'n'drop)
- onkeypress: ao pressionar e soltar uma tecla
- onkeydown: ao pressionar uma tecla.
- onkeyup: ao soltar uma tecla. Mesmo acima.
- onblur: quando um elemento perde foco
- onfocus: quando um elemento ganha foco
- onchange: quando um input, select ou textarea tem seu valor alterado
- onload: quando a página é carregada

- onunload: quando a página é fechada
- onsubmit: disparado antes de submeter o formulário. Útil para realizar validações

Existe também uma série de outros eventos mais avançados que permitem a criação de interações para drag-and-drop, e até mesmo a criação de eventos customizados.

ARRAY

O array é útil quando precisamos trabalhar com diversos valores armazenados:

```
var palavras = ["Instituição", "Ensino"];
palavras.push("Inovação"); // adiciona a string "Inovação"
```

Também é possível guardar valores de tipos diferentes:

```
var variosTipos = ["Instituição", 10, [1,2]];
```

Como obter um valor agora? Lembre-se que o tamanho de um array vai de 0 até o seu tamanho - 1.

```
console.log(variosTipos[1]) // imprime 15!
```

Outros aspectos interessantes é o tamanho do array: podemos adicionar quantos elementos quisermos que seu tamanho aumentará quando necessário.

BLOCOS DE REPETIÇÃO

Muitas vezes precisamos executar um trecho de código repetidamente até que uma condição seja contemplada, ou enquanto uma condição for verdadeira. Para isso, o JavaScript oferece uma série de blocos de repetição. O mais comum é o for.

For

O bloco for precisa de algumas informações de controle para evitar que ele execute infinitamente:

```
for (/* variável de controle */; /* condição */; /* pós execução */) {
// código a ser repetido
```

```
}
```

Das informações necessárias, somente a condição é obrigatória, mas normalmente utilizamos todas as informações:

```
var palavras = ["Instituição", "Ensino"];
for (var i = 0; i < palavras.length; i++) {
alert(palavras[i]);
}
```

While

O bloco while executa determinado código repetitivamente enquanto uma condição for verdadeira. Diferente do bloco for, a variável de controle, bem como sua manipulação, não são responsabilidades do bloco em si:

```
var contador = 1;
while (contador <= 10) {
alert(contador + " Ovelha...");
contador++;
}
alert("Valor do contador: " + contador);
```

FUNÇÕES TEMPORAIS

Em JavaScript, podemos criar um timer para executar um trecho de código após um certo tempo, ou ainda executar algo de tempos em tempos. A função setTimeout permite que agendemos alguma função para execução no futuro e recebe o nome da função a ser executada e o número de milissegundos a esperar:

```
// executa a minhaFuncao daqui um segundo
setTimeout(minhaFuncao, 1000);
```

Se for um código recorrente, podemos usar o setInterval que recebe os mesmos argumentos mas executa a função indefinidamente de tempos em tempos:

```
// executa a minhaFuncao de um em um segundo
setInterval(minhaFuncao, 1000);
```

Framework jQuery

Jquery é uma biblioteca que traz diversas funcionalidades voltadas à solução dos problemas mais difíceis de serem contornados com o uso do JavaScript puro.

A principal vantagem na adoção de uma biblioteca de JavaScript é permitir uma maior compatibilidade de um mesmo código com diversos navegadores. Uma maneira de se atingir esse objetivo é criando funções que verificam quaisquer características necessárias e permitam que o programador escreva um código único para todos os navegadores.

Além dessa vantagem, o jQuery, que é hoje a biblioteca padrão na programação para Web, traz uma sintaxe mais "fluida" nas tarefas mais comuns ao programador que são: selecionar um elemento do documento e alterar suas características.

JQUERY - A FUNÇÃO $

O jQuery é uma grande biblioteca que contém diversas funções que facilitam a vida do programador. A mais importante delas, que inicia a maioria dos códigos, é a função $.

Com ela é possível selecionar elementos com maior facilidade, maior compatibilidade, e com menos código. Por exemplo:

```
//Utilizando apenas javascript
var cabecalho = document.getElementById("cabecalho");
if (cabecalho.attachEvent) {
cabecalho.attachEvent("onclick", function (event) {
alert("Você clicou no cabeçalho, usuário do IE!");
});
} else if (cabecalho.addEventListener) {
cabecalho.addEventListener("click", function (event) {
alert("Você clicou no cabeçalho!")
}, false);
}
```

```
//Utilizando jQuery
$("#cabecalho").click(function (event) {
alert("Você clicou no cabeçalho!");
});
```

Note como a sintaxe do jQuery é bem menor, e a biblioteca se encarrega de encontrar o modo mais compatível possível para adicionar o evento ao elemento cujo id é cabeçalho.

Existem diversas funções que o jQuery permite que utilizemos para alterar os elementos que selecionamos pela função $, e essas funções podem ser encadeadas, por exemplo:

```
$("#cabecalho").css({"margin-top": "20px", "color": "#333333"}).addClass("selecionado");
```

No código acima, primeiramente chamamos a função $ e passamos como argumento uma String idêntica ao seletor CSS que utilizaríamos para selecionar o elemento de id cabeçalho. Na sequência chamamos a função css e passamos um objeto como argumento, essa função adicionará ou alterará as informações desse objeto como propriedades de estilo do elemento que selecionamos com a função $.

Em seguida chamamos mais uma função, a addClass, que vai adicionar o valor "selecionado" ao atributo class do elemento com o id "cabeçalho". Dessa maneira, é possível fazer muito mais com muito menos código, e ainda por cima de uma maneira que funciona em diversos navegadores.

JQUERY SELECTORS

Um dos maiores poderes do jQuery está na sua capacidade de selecionar elementos a partir de seletores CSS.

Como já aprendemos, exitem diversas formas de selecionarmos quais elementos ganharão determinado estilo. Infelizmente muitos desses seletores não funcionam em todos os navegadores. Contudo, no jQuery, os temos todos à nossa disposição.

Por exemplo, se quisermos esconder todas as tags <td> filhas de um <tbody>, basta:

```
$('tbody td').hide();
```

Seletores mais comuns:

```
// pinta o fundo do formulario com id "form" de preto
$('#form').css('background', 'black');
// esconde todos os elementos com o atributo "class" igual a "headline"
$('.headline').hide();
// muda o texto de todos os parágrafos
$('p').text('olá mundo');
```

Mais exemplos:

```
$('div > p:first'); // o primeiro elemento <p> imediatamente filho de um
<div>
$('input:hidden'); // todos os inputs invisíveis
$('input:selected'); // todas as checkboxes selecionadas
$('input[type=button]'); // todos os inputs com type="button"
$('td, th'); // todas as tds e ths
```

Lembre-se de que a função que chamamos após o seletor é aplicada para todos os elementos retornados. Veja:

```
// forma ineficiente
alert($('div').text() + $('p').text() + $('ul li').text());
// forma eficiente :D
alert($('div, p, ul li').text());
```

A função text() é chamada para todos os s, <p>s, e s filhos de s.

FILTROS CUSTOMIZADOS E POR DOM

Existem diversos seletores herdados do css que servem para selecionar elementos baseados no DOM. Alguns deles são:

```
$('div > p'); // <p>s imediatamente filhos de <div>
$('p + p'); // <p>s imediatamente precedidos por outro <p>
$('div:first-child'); // um elemento <div> que seja o primeiro filho
$('div:last-child'); // um elemento <div> que seja o último filho
$('div > *:first-child'); // um elemento que seja o primeiro filho direto de uma <div>
$('div > *:last-child'); // um elemento que seja o ultimo filho direto de uma <div>
$('div p:nth(0)'); // o primeiro elemento <p> filho de uma <div>
$('div:empty'); // <div>s vazios
```

UTILITÁRIO DE ITERAÇÃO DO JQUERY

O jQuery traz também entre suas diversas funcionalidades, uma função que facilita a iteração em elementos de um Array com uma sintaxe mais agradável:

```
$("#menu-departamentos li").each(function (index, item) {
alert(item.text());
});
```

A função each chamada logo após um seletor executa a função que passamos como argumento para cada um dos itens encontrados. Essa função precisa de dois argumentos. O primeiro será o "índice" do elemento atual na coleção (0 para o primeiro, 1 para o segundo e assim por diante), e o segundo será o próprio elemento.

Também é possível utilizar a função each do jQuery com qualquer Array:

```
var pessoas = ["João", "José", "Maria", "Antônio"];
$.each(pessoas, function(index, item) {
alert(item);
})
```

Nesse caso, chamamos a função each diretamente após o $, pois essa implementação é um método do próprio objeto $. Passamos dois argumentos, o primeiro é o Array que queremos percorrer e o segundo a função que desejamos executar para cada um dos itens do Array.

CARACTERÍSTICAS DE EXECUÇÃO

Para utilizarmos o jQuery em nossos projetos com maior segurança, devemos tomar alguns cuidados.

Importação

Antes de mais nada é necessário incluir o jQuery em nossa página. Só assim o navegador executará seu código para que possamos utilizar suas funcionalidades em nosso código.

Por isso é necessário que a tag <script> do jQuery seja a primeira de todas na ordem de nosso documento:

```
<script src="scripts/jquery.js"></script>
<!-- só podemos utilizar o jQuery após sua importação -->
<script src="scripts/meuscript.js"></script>
<script src="scripts/meuoutroscript.js"</script>
```

Executar somente após carregar

Como estamos constantemente selecionando elementos do documento e alterando suas características, é importante garantir que os elementos que pretendemos utilizar já tenham sido carregados pelo navegador. A melhor maneira de garantir isso é somente executar nosso script após o término do carregamento total da página com a função $ dessa maneira:

```
$(function() {
$("#cabecalho").css({"background-color": "#000000"});
})
```

Essa função $ que recebe uma função anônima como argumento garante que o código dentro dela só será executado ao fim do carregamento de todos os elementos da página.

PLUGINS JQUERY

Além de usar os componentes JavaScript que vêm prontos no Bootstrap, podemos baixar outros prontos. São plugins feitos para o jQuery ou para o Bootstrap que trazem novas funcionalidades.

A grande riqueza do jQuery é justo sua vasta gama de plugins disponíveis. Há até um diretório no site deles: http://plugins.jquery.com/

Cada plugin é um arquivo JavaScript que você inclui na página e adiciona uma funcionalidade específica. Muitos exigem que escrevamos um pouco de código para configurar seu uso; outros são mais plug and play.

Você vai precisar consultar a documentação do plugin específico quando for usar.

PHP

ORIENTAÇÃO A OBJETO

Introdução

Neste módulo vamos abordar a orientação a objetos, seus diversos conceitos e técnicas das mais diversas formas, sempre com um exemplo ilustrado de código-fonte com aplicabilidade prática.

Orientação a objetos

Ao trabalharmos com orientação a objetos é fundamental entender o conceito de classes e objetos. Uma classe é uma estrutura que define um tipo de dados, podendo conter atributos (variáveis) e também funções (métodos) para manipular esses atributos.

No exemplo a seguir, declaramos a classe Produto com quatro propriedades, ou seja, quatro "variáveis" que existem dentro do contexto do objeto. Declaramos propriedades por meio da palavra-chave var, dentre formas vistas a seguir.

produto.php

```
<?php
class Produto
{
var $codigo;
var $descricao;
var $preco;
var $quantidade;
}
?>
```

Um objeto contém exatamente a mesma estrutura e as propriedades de uma classe, no entanto sua estrutura é dinâmica, seus atributos podem mudar de valor durante a execução do programa e podemos declarar diversos objetos oriundos de uma mesma classe.

No exemplo a seguir, estamos fazendo uso da classe Produto, criando um objeto (produto). Para instanciar um objeto é utilizado o operador new, seguido do nome da classe. Para acessar propriedades de um objeto em nosso programa, utilizamos o nome da propriedade precedido do nome do objeto.

Um objeto não foi feito para ser impresso diretamente, mas suas propriedades sim. Entretanto, se tentarmos imprimir um objeto diretamente (como na última linha do programa a seguir), o PHP retornará o identificador interno deste objeto na memória. Mesmo que criássemos vários objetos de uma mesma classe, cada um deles possuiria um identificador próprio e um comportamento único, diferenciando uns dos outros.

objeto.php

```
<?php
//insere a classe
include_once 'produto.php';


// cria um objeto
$produto = new Produto;
```

```
// atribuir valores
$produto->codigo = 4001;
$produto->descricao = 'CD - The Best of Eric Clapton';
echo $produto;
?>
```

Resultado

```
Object id #1
```

Reescreveremos, então, a classe Produto para adicionar uma funcionalidade ou um método, que é uma função declarada dentro da estrutura da classe, agindo dentro desse escopo.

O método criado, imprimeEtiqueta(), trabalha com as propriedades do objeto. A fim de diferenciar as propriedades de um objeto de variáveis locais, utiliza-se a pseudovariável $this para representar o objeto atual e acessar suas propriedades.

produto.php

```
<?php
class Produto
{

var $codigo;
var $descricao;
var $preco;
var $quantidade;

function imprimeEtiqueta()
{
print 'Código: ' . $this->codigo . "\n";
print 'Descrição: ' . $this->descricao . "\n";
}

}
?>
```

Reescreveremos, então, o exemplo anterior para demonstrar a utilização do método imprimeEtiqueta() e criaremos dois objetos para demonstrar tal característica.

objeto.php

```
<?php
// insere a classe
include_once 'Produto.php';

// cria dois objetos
$produto1 = new Produto;
$produto2 = new Produto;

// atribuir valores
$produto1->codigo = 4001;
$produto1->descricao = 'CD - The Best of Eric Clapton';

$produto2->codigo = 4002;
$produto2->descricao = 'CD - The Eagles Hotel California';

// imprime informações de etiqueta
$produto1->imprimeEtiqueta();
$produto2->imprimeEtiqueta();
?>
```

Resultado

```
Código: 4001
Descrição: CD - The Best of Eric Clapton
Código: 4002
Descrição: CD - The Eagles Hotel California
```

Classe

A classe é uma estrutura estática utilizada para descrever objetos mediante atributos (propriedades) e métodos (funcionalidades). A classe é um modelo ou template para criação desses objetos. Tem-se por propriedades características intrínsecas à classe em questão.

Podem ser classes: entidades do negócio da aplicação (pessoa, conta, cliente, fornecedor), entidades de interface (janela, botão, painel, frame, barra), dentre outras (conexão com banco de dados, um arquivo-texto, um arquivo XML, uma conexão SSH, uma conexão FTP, um Web Service).

No momento em que modelamos uma classe Pessoa, são suas propriedades: nome, altura, idade, nascimento, escolaridade e salario. Tem-se por métodos funcionalidades desempenhadas pela classe. No caso desta classe poderiam ser os métodos: crescer(), formar() e envelhecer().

Ao modelar uma classe ContaBancaria, são suas propriedades: agencia, codigo, dataDeCriacao, titular, senha, saldo, bem como são seus métodos (funcionalidades): retirar(), depositar() e obterSaldo().

As classes são orientadas ao assunto, ou seja, cada classe é responsável por um assunto diferente e possui responsabilidade sobre o mesmo. Ela deve proteger o acesso ao seu conteúdo por meio de mecanismos como o de encapsulamento (visto a seguir).

Dessa forma, criamos sistemas mais confiáveis e robustos. A classe Pessoa e a classe ContaBancaria são responsáveis pelas propriedades nelas contidas, evitando que essas propriedades contenham valores inconsistentes ou sejam manipuladas indevidamente.

Os membros de uma classe são declarados na ordem: primeiro as propriedades (mediante o operador var) e, em seguida, os métodos (pelo operador function, também utilizado para declarar funções).

As classes Pessoa e ContaBancaria são apresentadas a seguir:

pessoa.php

```
<?php
class Pessoa
{

var $codigo;
var $nome;
var $altura;
var $idade;
var $nascimento;
var $escolaridade;
var $salario;

/***
* método crescer
* aumenta a altura em centímetros
*/

function crescer($centimetros)
{
if ($centimetros > 0)
```

```
{
$this->altura += $centimetros;
}
}

/***
* método estudar
* altera a escolaridade para $titulacao
*/
function formar($titulacao)
{
$this->escolaridade = $titulacao;
}

/***
* método envelhecer
* aumenta a idade em $anos
*/
function envelhecer($anos)
{
if ($anos > 0)
{
$this->idade += $anos;
}
}

}
?>
```

conta_bancaria.php

```
<?php
class ContaBancaria
{
var $agencia;
var $codigo;
var $dataDeCriacao;
```

```
var $titular;
var $senha;
var $saldo;

/***
* método retirar
* diminui o saldo em $quantia
*/
function retirar($quantia)
{
if ($quantia > 0)
{
$this->saldo -= $quantia;
}
}

/***
* método depositar
* aumenta o saldo em $quantia
*/
function depositar($quantia)
{
if ($quantia > 0)
{
$this->saldo += $quantia;
}
}

/***
* método obterSaldo
* retorna o saldo atual
*/
function obterSaldo()
{
return Sthis->saldo;
}
```

```
}
?>
```

Note que dentro de uma classe, para referenciar um de seus membros (propriedades), basta utilizar-se da expressão $this, que nada mais é do que o nome de uma pseudovariável interna que representa o próprio objeto que estará sendo manipulado.

Objeto

Um objeto é uma estrutura dinâmica originada com base em uma classe. Após a utilização de uma classe para criar diversas estruturas iguais a ela, que interagem no sistema e possuem dados nela armazenados, dizemos que estamos criando objetos, ou mesmo instanciando objetos de uma classe.

Diz-se que o objeto é uma instância de uma classe, porque o objeto existe durante um dado instante de tempo - da sua criação até a sua destruição.

São objetos da classe Pessoa: Carlos, João, Maria e José.

Todos têm uma estrutura igual (como a da classe Pessoa), mas propriedades com valores diferentes, caracterizando cada um de forma distinta, dando-lhes unicidade no sistema.

Para instanciar um objeto de uma determinada classe, procedemos para a declaração de uma variável qualquer (nosso objeto em questão) e lhe atribuímos o operador new seguido do nome da classe que desejamos instanciar.

Veja a seguir um exemplo da utilização de objetos.

objetos.php

```
<?php
# carrega as classes
include_once 'pessoa.php' ;
include_once 'conta_bancaria.php' ;

# criação do objeto $carlos

$carlos = new Pessoa;
$carlos->codigo = 10;
$carlos->nome = 'Carlos da Silva';
$carlos->altura = 1.85;
$carlos->idade = 25;
$carlos->nascimento = '10/04/1976';
```

```
$carlos->escolaridade = 'Ensino Médio';

echo "Manipulando o objeto $carlos->nome :\n";
echo "{$carlos->nome} é formado em: {$carlos->escolaridade} \n";
$carlos->formar('Técnico em Eletricidade');
echo "{$carlos->nome} é formado em: {$carlos->escolaridade} \n";
echo "{$carlos->nome} possui {$carlos->idade} anos \n";
$carlos->Envelhecer(1);
echo "{$carlos->nome} possui {$carlos->idade} anos \n";

# criação do objeto $conta_carlos
$conta_carlos = new Conta;
$conta_carlos->agencia = 6677;
$conta_carlos->codigo = 'CC.1234.56';
$conta_carlos->dataDeCriacao = '10/07/02';
$conta_carlos->titular = $carlos;
$conta_carlos->senha = 9876;
$conta_carlos->saldo = 567.89;

echo "\n";
echo "Manipulando a conta de: {$conta_carlos->titular->nome} \n";
echo "0 saldo atual é R\$ { $conta_carlos->obterSaldo() } \n";
$conta_carlos->depositar(20);
echo "0 saldo atual é R\$ { $conta_carlos->obterSaldo() } \n";
$conta_carlos->retirar(10);
echo "0 saldo atual é R\$ { $conta_carlos->obterSaldo() } \n";
?>
```

Resultado

```
Manipulando o objeto Carlos da Silva :
Carlos da Silva é formado em: Ensino Médio
Carlos da Silva é formado em: Técnico em Eletricidade
Carlos da Silva possui 25 anos
Carlos da Silva possui 26 anos
Manipulando a conta de: Carlos da Silva
0 saldo atual é R$ 567.89
0 saldo atual é R$ 587.89
```

```
0 saldo atual é R$ 577.89
```

Observação: note que, para acessar propriedades e métodos de um objeto dentro de uma string dupla (que é interpretada), é necessário utilizar-se de chaves ao redor da expressão.

Construtores e destrutores

Um construtor é um método especial utilizado para definir o comportamento inicial de um objeto, ou seja, o comportamento no momento de sua criação.

O método construtor é executado automaticamente no momento em que instanciamos um objeto por meio do operador new. Assim, não devemos retornar nenhum valor por meio do método construtor porque o mesmo retorna por definição o próprio objeto que está sendo instanciado.

Caso não seja definido um método construtor, automaticamente todas as propriedades do objeto criado são inicializadas com o valor NULL. Nos casos mostrados anteriormente (Pessoa e ContaBancaria), há a necessidade de um método construtor para definir os valores iniciais para as suas propriedades.

Para definir um método construtor em uma determinada classe basta declarar o método __construct().

Um destrutor ou finalizador é um método especial executado automaticamente quando o objeto é desalocado da memória, quando atribuímos o valor NULL ao objeto, quando utilizamos a função unset sobre o mesmo ou, em última instância, quando o programa é finalizado.

O método destrutor pode ser utilizado para finalizar conexões, apagar arquivos temporários criados durante o ciclo de vida do objeto, dentre outras circunstâncias.

Para definir um método destrutor em uma determinada classe basta declarar o método __destruct().

A seguir, complementaremos as classes Pessoa e ContaBancaria adicionando os métodos construtores e destrutores.

Nesta etapa suprimiremos parte do código criada anteriormente.

pessoa.php (trecho adicional)

```
<?php
class Pessoa
{

# ...
```

```
# conteúdo já escrito no exemplo anterior
# ...

/***
* método construtor
* inicializa propriedades
*/
function __construct($codigo, $nome, $altura, $idade, $nascimento, $escolaridade, $salario)
{
$this->codigo = $codigo;
$this->nome = $nome;
$this->altura = $altura;
$this->idade = $idade;
$this->nascimento = $nascimento;
$this->escolaridade = $escolaridade;
$this->salario = $salario;
}

/***
* método destrutor
* finaliza Objeto
*/
function __destruct()
{
echo "Objeto {$this->nome} finalizado...\n ";
}
}
?>
```

conta_bancaria. php (trecho adicional)

```
<?php
class contaBancaria
{

# ...
# conteúdo já escrito no exemplo anterior
```

```
# . . .

/***
* método construtor
* inicializa propriedades
*/
function __construct($agencia, $codigo, $dataDeCriacao, $titular, $senha, $saldo)
{

$this->agencia = $agencia;
$this->codigo = $codigo;
$this->dataDeCriacao = $dataDeCriacao;
$this->titular = $titular;
$this->senha = $senha;

// chamada a outro método da classe
$this->depositar($saldo);
}

/***
* método destrutor
* finaliza objeto
*/
function __destruct()
{
echo "Objeto Conta {$this->codigo} de {$this->titular->nome} finalizada...\n";
}
}
?>
```

construtores.php

```
<?php
# carrega as classes
include_once 'pessoa.php';
include_once 'conta_bancaria.php';
```

```
# criação do objeto $carlos
$carlos = new Pessoa(10,'Carlos da Silva',1.85, 25,'10/04/1976','Ensino Médio',650);

echo "Manipulando o objeto {$carlos->nome}: \n";
echo "{$carlos->nome} é formado em: {$carlos->escolaridade} \n";
$carlos->formar('Técnico em Eletricidade');
echo "{$carlos->nome} é formado em: {$carlos->escolaridade} \n";
echo "{$carlos->nome} possui {Scarlos->idade} anos \n";
$carlos->envelhecer(1);
echo "{$carlos->nome} possui {$carlos->idade} anos \n";

# Criação do objeto $conta_carlos
$conta_carlos = new Conta(6677, 'CC.1234.56', '10/07/02', $carlos, 9876, 567.89);
echo "\n";
echo "Manipulando a conta de: { $conta_carlos->titular->nome }: \n";
echo "0 saldo atual é R\$ { $conta_carlos->obterSaldo() } \n";
$conta_carlos->depositar(20);
echo "0 saldo atual é R\$ { $conta_carlos->obterSaldo() } \n";
$conta_carlos->retirar(10);
echo "0 saldo atual é R\$ { $conta_carlos->obterSaldo() } \n";
?>
```

Resultado:

```
Manipulando o objeto Carlos da Silva:
Carlos da Silva é formado em: Ensino Médio
Carlos da Silva é formado em: Técnico em Eletricidade
Carlos da Silva possui 25 anos
Carlos da Silva possui 26 anos
Manipulando a conta de: Carlos da Silva:
0 saldo atual é R$ 567.89
0 saldo atual é R$ 587.89
0 saldo atual é R$ 577.89
Objeto Carlos da Silva finalizado...
Objeto Conta CC.1234.56 de Carlos da Silva finalizada...
```

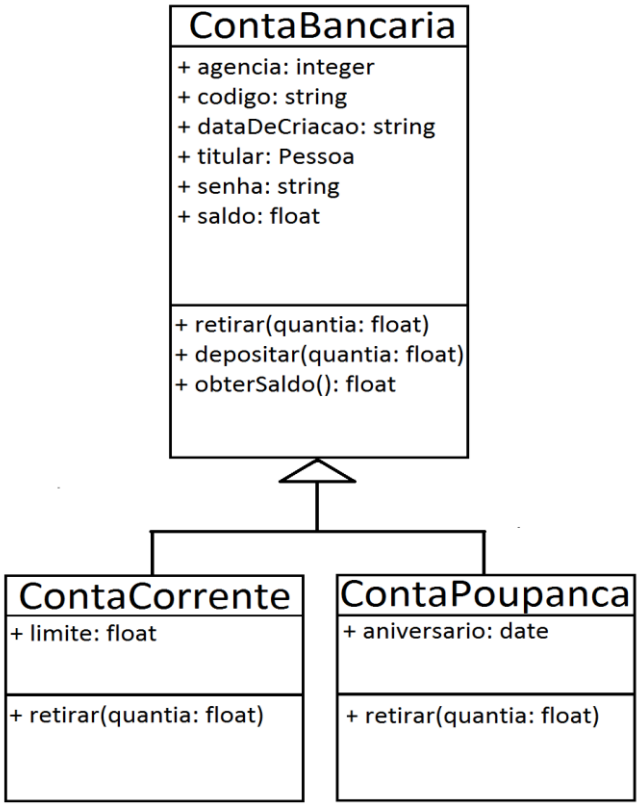
Herança

A utilização da orientação a objetos e do encapsulamento do código em classes nos orienta em direção a uma maior organização, mas um dos maiores benefícios que encontramos na utilização desse paradigma é o reuso. A possibilidade de reutilizar partes de código já definidas é o que nos dá maior agilidade no dia-a-dia, além de eliminar a necessidade de eventuais duplicações ou reescritas de código.

Quando falamos em herança, a primeira imagem que nos aparece na memória é a de uma árvore genealógica com avós, pais, filhos e nas características que são transmitidas geração após geração. O que devemos levar em consideração sobre herança em orientação a objetos é o compartilhamento de atributos e comportamentos entre as classes de uma mesma hierarquia (árvore). As classes inferiores da hierarquia automaticamente herdam todas as propriedades e os métodos das classes superiores, chamadas de superclasses.

Este recurso tem uma aplicabilidade muito grande, visto que é relativamente comum termos de criar novas funcionalidades em software. Utilizando a herança, em vez de criarmos uma estrutura totalmente nova (uma classe), podemos reaproveitar uma estrutura já existente que nos forneça uma base abstrata para o desenvolvimento, provendo recursos básicos e comuns.

Já criamos a classe genérica ContaBancaria; agora podemos aproveitar seu código-fonte para criar classes mais específicas como ContaCorrente e ContaPoupanca, como no diagrama a seguir:



Veja, a seguir, como fica o código das classes-filha utilizando o mecanismo de herança. Nestes exemplos, vemos a ocorrência do fenômeno chamado sobrescrita (overriding), que acontece quando modificamos o comportamento de um método da classe-pai (referida pelo operador parent) na classe-filha, adicionado a ela uma nova funcionalidade. Assim, na construção da classe ContaPoupanca adicionamos a inicialização da propriedade aniversario e, na classe ContaCorrente, adicionamos a inicialização da propriedade limite. O método retirar() também foi sobrescrito. Na classe ContaPoupanca é verificado se há saldo para realizar a transação e na classe ContaCorrente é verificado se a retirada está dentro do limite da conta, além de debitar CPMF.

conta_poupanca.php

```
<?php
class ContaPoupanca extends ContaBancaria
{
var $aniversario;
/***
*  método construtor (sobrescrito)
* agora, também inicializa a variável $aniversario
*/
function __construct($agencia,$codigo,$dataDeCriacao,$titular,
$senha,$saldo,$aniversario)
{
// chamada do método construtor da classe-pai.
parent::
__construct($agencia,$codigo,$dataDeCriacao,$titular,$senha,$saldo);
$this->aniversario = $aniversario;
}

/***
* método retirar (sobrescrito)
* verifica se há saldo para retirar tal $quantia.
*/
function retirar($quantia)
{
i f ($this->saldo >= $quantia)
{
// Executa método da classe-pai.
parent::retirar($quantia);
// retirada permitida
return true;
```

```
}
else
{
echo "Retirada não permitida...\n";
return false;
}
}

}
?>
```

conta_corrente.php

```
<?php
class ContaCorrente extends ContaBancaria
{

var $limite;

/***
* método construtor (sobrescrito)
* agora, também inicializa a variável $limite
*/
function construct($agencia, $codigo, $dataDeCriacao, $titular, $senha, $saldo, $limite)
{
// chamada do método construtor da classe-pai.
parent:: __construct($agencia, $codigo, $dataDeCriacao, $titular, $senha, $saldo);
$this->limite = $limite;
}

/***
* método retirar (sobrescrito)
* verifica se a $quantia retirada está dentro do limite.
*/
function retirar($quantia)
{
```

```
// imposto sobre movimentação financeira
$cpmf = 0.05;
if ( ($this->saldo + $this->limite) >= $quantia )
{
// Executa método da classe-pai.
parent::retirar($quantia);

// Debita o Imposto
parent::retirar($quantia * $cpmf);

// retirada permitida
return true;
}
else
{
echo "Retirada não permitida...\n";
return false;
}
}

}
?>
```

Polimorfismo

O significado da palavra polimorfismo nos remete a "muitas formas". Polimorfismo em orientação a objetos é o princípio que permite que classes derivadas de uma mesma superclasse tenham métodos iguais (com a mesma nomenclatura e parâmetros), mas comportamentos diferentes, redefinidos em cada uma das classes-filha.

Veja que as classes ContaPoupanca e ContaCorrente, criadas anteriormente, possuem os mesmos métodos. Repare que o método retirar() possui o mesmo nome, mas comportamento diferente em ambas. No caso da conta poupança, ele verifica somente se há saldo. Na conta corrente, verifica se a retirada está dentro do limite da operação, além de debitar o imposto correspondente (CPMF). Veja que o comportamento dessas operações é muito similar.

Inicializamos duas contas com os mesmos valores e efetuamos os mesmos procedimentos sobre elas dentro de um laço de repetição. No exemplo a seguir, em vez de armazenar os

objetos do tipo conta em variáveis, estamos formando um array de objetos: $contas. Veja os resultados obtidos.

polimorfismo.php

```
<?php

# carrega as classes
include_once 'pessoa.php';
include_once 'conta_bancaria.php';
include_once 'conta_poupanca. php';
include_once 'conta_corrente.php';

# Criação do objeto $carlos
$carlos = new Pessoa(10, 'Carlos da Silva', 1.85, 25, '10/04/1976', 'Ensino Médio', 650.00);
echo "Manipulando o objeto {Scarlos->nome}: \n";

# Criação do objeto $conta_carlos
$contas[1] = new ContaCorrente(6677, 'CC.1234.56','10/07/02',$carlos,9876,500.00,200.00);

$contas[2] = new ContaPoupanca(6678, 'PP.1234.57', '10/07/02',$carlos,9876,500.00,'10/07');

// percorremos as contas
foreach ($contas as $key => $conta)
{
echo "Manipulando a conta $key de: { $conta->titular->nome }: \n";
echo "0 saldo atual da conta $key é R\$ { $conta->obterSaldo() } \n";
$conta->depositar(200);
echo "0 saldo atual da conta $key é R\$ { $conta->obterSaldo() } \n";
$conta->retirar(100);
echo "0 saldo atual da conta $key é R\$ { $conta->obterSaldo() } \n";
}

?>
```

Resultado:

```
Manipulando o objeto Carlos da Silva:
Manipulando a conta 1 de: Carlos da Silva:
```

```
0 saldo atual da conta 1 é RS 500
0 saldo atual da conta 1 é RS 700
0 saldo atual da conta 1 é RS 595
Manipulando a conta 2 de: Carlos da Silva:
0 saldo atual da conta 2 é R$ 500
0 saldo atual da conta 2 é R$ 700
0 saldo atual da conta 2 é R$ 600
Objeto Carlos da Silva finalizado...
Objeto Conta CC.1234.56 de Carlos da Silva finalizada...
Objeto Conta PP.1234.57 de Carlos da Silva finalizada...
```

Observação: o PHP não suporta sobrecarga, ou métodos com o mesmo nome, mas assinaturas (parametrização) diferentes.

Abstração

No paradigma de orientação a objetos se prega o conceito da "abstração". De acordo com o dicionário, "abstrair" é separar mentalmente, considerar isoladamente, simplificar, alhear-se. Para construir um sistema orientado a objetos, não devemos projetar o sistema como sendo uma grande peça monolítica; devemos separá-lo em partes, concentrando-nos nas peças mais importantes e ignorando os detalhes (em um primeiro momento), para podermos construir peças bem-definidas que possam ser reaproveitadas mais tarde, formando uma estrutura hierárquica.

Classes abstratas

Nesse contexto, encontraremos classes estruturais, ou seja, que estão na nossa hierarquia de classes para servirem de base para outras. São classes que nunca serão instanciadas na forma de objetos; somente suas filhas serão. Nestes casos, é interessante marcar essas classes como sendo classes abstratas, de modo que cada classe abstrata é tratada diferentemente pela linguagem de programação, a qual irá automaticamente impedir que se instanciem objetos a partir dela.

Seguindo os exemplos anteriores, uma pessoa pode ter uma ContaCorrente ou uma ContaPoupança, mas jamais poderá ter uma ContaBancaria. Isso porque ContaBancaria é uma estrutura abstrata, não definindo características próprias como taxas de retirada, limites, dentre outras especificidades que são escritas nas classes-filha.

No exemplo a seguir, tentaremos instanciar um objeto da classe ContaBancaria, mesmo que ela seja abstrata, e obteremos a mensagem de erro dizendo que a classe abstrata ContaBancaria não pode ser instanciada.

conta_bancaria.php (trecho adicionado)

```
<?php
abstract class ContaBancaria
{

# . . .
# conteúdo já escrito no exemplo anterior
# . . .

}
?>
```

classe_abstrata.php

```
<?php
Include_once 'conta_bancaria.php';
$conta = new ContaBancaria;
?>
```

Resultado:

```
Fatal error: Cannot instantiate abstract class ContaBancaria in classe_abstrata.php on line 3
```

Classes finais

A classe final não pode ser uma superclasse, ou seja, não pode ser base em uma estrutura de herança. Se definirmos uma classe como final pelo operador FINAL, ela não poderá mais ser especializada. Em nosso exemplo, tornamos a classe ContaPoupanca uma classe final e, mesmo assim, tentamos especializá-la no que chamamos de ContaPoupancaUniversitaria. A mensagem de erro obtida diz que a classe ContaPoupancaUniversitaria não pode descender da classe final ContaPoupanca.

conta_poupanca.php (trecho adicional)

```
<?php
final class ContaPoupanca extends ContaBancaria
{
```

```
# . . .
# Conteúdo já escrito no exemplo anterior
# . . .

}
?>
```

classe_final.php

```
<?php
include_once 'conta_bancaria.php';
include_once 'conta_poupanca.class.php';
class ContaPoupancaUniversitaria extends ContaPoupanca
{

//... sobrescrita de métodos

}
?>
```

Resultado:

```
Fatal error: Class ContaPoupancaUniversitaria may not inherit from final class
ContaPoupanca in classe_final.php on line 4
```

Métodos abstratos

Um método abstrato consiste na definição de uma assinatura na classe abstrata. Este método deverá conter uma implementação na classe-filha, mas não deve possuir implementação na classe em que ele é definido. Em nosso exemplo, definiremos um método abstrato na classe ContaBancaria. Isso faz com que seja obrigatório a qualquer classe descendente da classe ContaBancaria (vide ContaPoupanca e ContaCorrente) ter em si a implementação deste método.

No exemplo que segue, definiremos o método transferir() na classe ContaBancaria como sendo abstrato e tentaremos fazer uso das classes ContaBancaria e ContaPoupanca.

conta_bancaria.php (trecho adicional)

```
<?php
abstract class ContaBancaria
```

```
{
# . . .
# conteúdo já escrito no exemplo anterior
# . . .

abstract function transferir($Conta, $valor);
}
?>
```

metodo_abstrato.php

```
<?php
include_once 'pessoa. php';
include_once 'conta_bancaria.php';
include_once 'conta_poupanca. php';
$carlos = new Pessoa(10, "Carlos da Silva", 1.85, 25, 72,"Ensino Médio", 650.00);
$conta = new ContaPoupanca(6677, "CC.1234.56", "10/07/02", $carlos, 9876, 500.00, '10/07');
?>
```

Resultado:

```
Fatal error: Class ContaPoupanca contains 1 abstract methods and must therefore be declared abstract (ContaBancaria::transferir) in conta_poupanca.php
```

Como esperado, o resultado que obtivemos foi um erro, indicando que a classe ContaPoupanca não possui a implementação do método transferir(). Para sanar o erro, segue o complemento do código da classe ContaPoupanca. O complemento do código-fonte da classe ContaCorrente é apresentado no exemplo seguinte.

conta_poupanca. php (trecho adicional)

```
<?php
class ContaPoupanca extends ContaBancaria
{

# . . .
# conteúdo já escrito no exemplo anterior
# . . .

function fransferir($conta, $valor)
```

```
{
i f ($this->retirar($valor))
{
$conta->depositar($valor);
}
}

}
?>
```

Métodos finais

Um método final não pode ser sobrescrito, ou seja, não pode ser redefinido na classe-filha. Para marcar um método como final, basta utilizar o operador final no início da sua declaração.

No exemplo a seguir, declararemos o método transferir() da classe ContaCorrente como sendo final e, ainda assim, tentaremos redefini-lo na classe-filha ContaCorrenteEspecial. O resultado é a mensagem de erro dizendo que não podemos sobrescrever o método transferi r() da classe ContaCorrente.

conta_corrente. php (trecho adicional)

```
<?php
class ContaCorrente extends ContaBancaria
{
var $taxaTransferencia =2.5;

# ...
# conteúdo já escrito no exemplo anterior
# . . .

final function transferir($conta, $valor)
{

i f ($this->retirar($valor))
{
$conta->depositar($valor);
```

```
}

i f ($this->titular != $conta->titular)
{
$this->retirar($this->taxaTransferencia);
}

}

}
?>
```

metodo_final.php

```
<?php
# carrega as classes
include_once 'conta_bancaria.php';
include_once 'conta_corrente.php';

class ContaCorrenteEspecial extends ContaCorrente
{

function depositar($valor)
{
echo "sobrescrevendo método depositar.\n";
parent::depositar($valor);
}

function transferir($conta, $valor)
{
echo "sobrescrevendo método transferir.\n";
parent::transferir($conta, $valor);
}

}
?>
```

Resultado:

```
Fatal error: Cannot override final method ContaCorrente::transferir() in metodo_final.php on line 6
```

Encapsulamento

Um dos recursos mais interessantes na orientação a objetos é o encapsulamento, um mecanismo que provê proteção de acesso aos membros internos de um objeto. Lembre-se que uma classe possui responsabilidade sobre os atributos que contém. Dessa forma, existem certas propriedades de uma classe que devem ser tratadas exclusivamente por métodos dela mesma, que são implementações projetadas para manipular essas propriedades da forma correta. As propriedades nunca devem ser acessadas diretamente de fora do escopo de uma classe, pois dessa forma a classe não fornece mais garantias sobre os atributos que contém, perdendo, assim, a responsabilidade sobre eles.

Para atingir o encapsulamento, uma das formas é definindo a visibilidade das propriedades e dos métodos de um objeto. A visibilidade define a forma como essas propriedades devem ser acessadas. Existem três formas de acesso:

- private - Membros declarados como pri vate somente podem ser acessados dentro da própria classe em que foram declarados. Não poderão ser acessados a partir de classes descendentes nem a partir do programa que faz uso dessa classe (manipulando o objeto em si). Na UML, simbolizamos com um (-) em frente à propriedade.

- protected - Membros declarados como protected somente podem ser acessados dentro da própria classe em que foram declarados e a partir de classes descendentes, mas não poderão ser acessados a partir do programa que faz uso dessa classe (manipulando o objeto em si). Na UML, simbolizamos com um (#) em frente à propriedade.

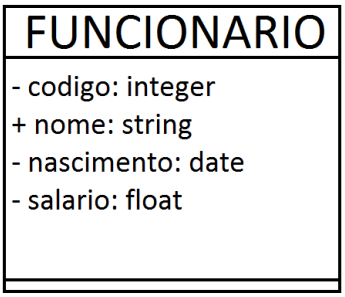
- public - Membros declarados como publ i c poderão ser acessados livremente a partir da própria classe em que foram declarados, a partir de classes descendentes e a partir do programa que faz uso dessa classe (manipulando o objeto em si). Na UML, simbolizamos com um (+) em frente à propriedade.

Até agora, nos exemplos anteriores, declaramos classes sem definir a visibilidade de propriedades e métodos. A visibilidade foi introduzida pelo PHP5 e, para manter compatibilidade com versões anteriores, quando a visibilidade de uma propriedade ou de um método não for definida, automaticamente ela será tratada como public.

Private

Para demonstrar a visibilidade private, criaremos a classe Funcionário e marcaremos a maioria das propriedades como private. Dessa forma, elas só poderão ser alteradas por métodos da mesma classe. A única propriedade que deixaremos para livre acesso será o nome, marcado como public.

Para demonstrar a necessidade de proteger o acesso a membros internos de uma classe, atribuiremos um valor inválido à propriedade Salario. Abaixo, temos a classe Funcionário no padrão UML.



funcionario.php

```
<?php
class Funcionario
{
private $codigo;
public $nome;
private $nascimento;
private $salario;
}
?>
```

private.php

```
<?php
# carrega a classe
include_once 'funcionario.php';
$pedro = new Funcionario;
$pedro->salario = 'Oitocentos e setenta e seis';
?>
```

Resultado:
Fatal error: Cannot access private property Funcionario::$salario in private.php on line 5

Veja que o PHP resulta em um erro de acesso à propriedade, como esperado, mas se não podemos alterar o valor da propriedade salario dessa forma, como faremos? Simples! Criaremos métodos pertencentes à classe Funcionario para manipular essa propriedade. No exemplo a seguir criaremos o método setSalario() para atribuir o valor da propriedade salario e getSalario() para retornar o valor. Aproveitamos para realizar uma validação no método setSalario(), que somente atribuirá o valor de salario caso este seja de um tipo numérico e positivo.

funcionario.php (trecho adicional)

```
<?php
class Funcionario
{
private $codigo;
public $nome;
private $nascimento;
private $salario;

/***
* método setSalario
* atribui o parâmetro $salario à propriedade $salario
*/
function setSalario($salario)
{
// verifica se é numérico e positivo
if (is_numeric($salario) and ($salario > 0))
{
$this->salario = $salario;
}
}

/***
* método getSalario
```

```
* retorna o valor da propriedade $salario
*/
function getSalario()
{
return $this->Salario;
}

}
?>
```

private.php

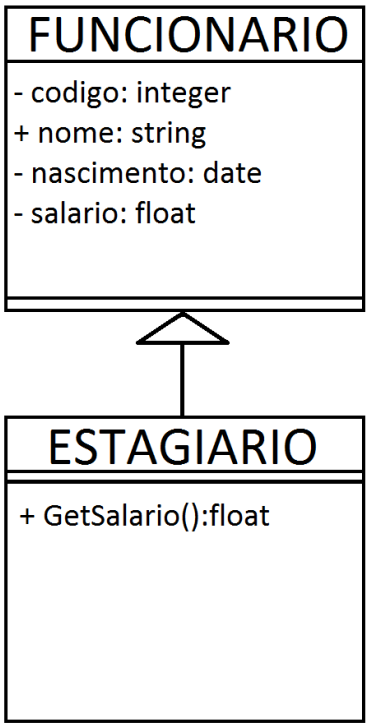
```
<?php
# carrega a classe
include_once 'funcionario.php' ;

// instancia novo Funcionário
$pedro = new Funcionario;

// atribui novo salário
$pedro->setSalario(876) ;

// obtém o Salário
echo 'Salário : (R$) ' . $pedro->getSalario();

?>
```

Resultado:

```
Salário : (R$) 876
```

Protected

Para demonstrar a visibilidade protected, vamos especializar a classe Funcionario, criando a classe Estagiario. A única característica exclusiva de um estagiário é que o seu salário é acrescido de 12% de bônus. Para tanto, sobrescreveremos o método getSalario() para que este retorne o salário já reajustado. Abaixo, temos o diagrama da classe Estagiario.



estagiario.php

```
<?php
class Estagiario extends Funcionario
{

/***
* método getSalario sobreescrito
* retorna o $salário com 12% de bônus.
*/
function getSalario()
{
return $this->salario * 1.12;
}

}
?>
```

protected.php

```
<?php
```

```
# carrega as classes
include 'funcionario.php';
include 'estagiario.php';

$pedrinho = new Estagiario;
$pedrinho->setSalario(248);
echo 'O Salário do Pedrinho é R$: ' . $pedrinho->getSalario() . "\n";
?>
```

Resultado:

```
O Salário do Pedrinho é R$: 0
```

Como esperado, o programa não retornou o salário esperado. Isso ocorre por que a propriedade salario é uma propriedade private, o que significa que ela somente pode ser acessada de dentro da classe em que ela foi declarada (Funcionario). Para permitir o acesso também nas classes-filha, alteraremos a classe Funcionario e marcaremos a propriedade salario como protected.

funcionario.php (trecho adicional)

```
<?php
class Funcionario
{

private $codigo;
public $nome;
private $nascimento;
protected $salario;

# ...
# conteúdo já escrito no exemplo anterior
# ...

}
?>
```

protected.php

```
<?php
# carrega as classes
include 'funcionario.php';
```

```
include 'estagiario. php';

$pedrinho = new Estagiario;
$pedrinho->setSalario(248);

echo 'O Salário do Pedrinho é R$: ' . $pedrinho->getSalario() . "\n";

?>
```

Resultado:

```
O Salário do Pedrinho é R$: 277.76
```

Agora, ao executarmos novamente nosso trecho de programa, veremos que o comportamento desejado é atingido, sendo a propriedade salario passível de alterações tanto no escopo da classe Funcionário quanto no escopo da classe Estagiario.

Public

Demonstrar a visibilidade public é uma tarefa simples, pois o comportamento padrão do PHP é tratar uma propriedade como public, ou seja, se não especificarmos a visibilidade, automaticamente ela será pública. No exemplo que segue, estamos criando dois objetos e alterando suas propriedades à vontade, as quais poderiam ser alteradas por métodos internos e por classes descendentes também.

public.php

```
<?php
# carrega as classes
include 'funcionario.php';
include 'estagiario.php';

// cria objeto Funcionario
$pedrinho = new Funcionario;
$pedrinho->nome = 'Pedrinho';

// cria objeto Estagiario
$mariana = new Estagiario;
$mariana->nome = 'Mariana';

// imprime propriedade nome
```

```
echo $pedrinho->nome;
echo $mariana->nome;

?>
```

Resultado:

```
Pedrinho
Mariana
```

Membros da classe

Como visto anteriormente, a classe é uma estrutura-padrão para criação dos objetos. A classe permite que armazenemos valores nela de duas formas: constantes de classe e propriedades estáticas. Estes atributos são comuns a todos os objetos da mesma classe.

Constantes

No exemplo a seguir, veremos como se dá a declaração de uma constante pelo operador const, seu acesso de forma externa ao contexto da classe, pela sintaxe NomeDaClasse::NomeDaConstante, e dentro da classe, pela sintaxe self: \NomeDaConstante. O operador self representa a própria classe.

constantes.php

```
<?php
class Biblioteca
{
        const Nome = "Sistema ";
}

class Aplicacao extends Biblioteca
{
        // declaração das constantes
        const Ambiente = "Maestro ";
        const Versão = "1.0.0.1";

        /* método construtor
         * acessa as constantes internamente
```

```
         */
        function __construct($Nome)
        {
                echo parent::Nome . self::Ambiente . self::Versao . $Nome . "\n";
        }
}

// acessa as constantes externamente
echo Biblioteca::Nome . Aplicacao:-.Ambiente . Aplicacao: :Versao . "\n";

new Aplicacao(' 2016');
new Aplicacao(' 2017');

?>
```

Resultado:

```
Sistema Maestro 1.0.0.1
Sistema Maestro 1.0.0.1 2016
Sistema Maestro 1.0.0.1 2017
```

Propriedades estáticas

Propriedades estáticas são atributos de uma classe; são dinâmicas como as propriedades de um objeto, mas estão relacionadas à classe. Como a classe é a estrutura comum a todos os objetos dela derivados, propriedades estáticas são compartilhadas entre todos os objetos de uma mesma classe.

propriedades_estaticas.php

```
<?php
class Aplicacao
{
        static $quantidade;

        /* método Construtor
         * incrementa a $quantidade de Aplicações
         */
        function __construct($nome)
        {
```

```
                // incrementa propriedade estática
                self::$quantidade ++;
                $i = self::$quantidade ;
                echo "Nova Aplicação nr. $i: $nome\n";
        }
}

# cria novos objetos
new Aplicacao('Básico');
new Aplicacao('Profissional');
new Aplicacao('Negocios');

echo 'Quantidade de Aplicações = ' . Aplicacao::$quantidade . "\n";

?>
```

Resultado:

```
Nova Aplicação nr. 1: Básico
Nova Aplicação nr. 2: Profissional
Nova Aplicação nr. 3: Negócios
Quantidade de Aplicações = 3
```

Métodos estáticos

Métodos estáticos podem ser invocados diretamente da classe, sem a necessidade de instanciar um objeto para isso. Eles não devem referenciar propriedades internas pelo operador $this, porque este operador é utilizado para referenciar instâncias da classe (objetos), mas não a própria classe; são limitados a chamarem outros métodos estáticos da classe ou utilizar apenas propriedades estáticas. Para executar um método estático, basta utilizar a sintaxe NomeDaClasse::NomeDoMétodo().

No exemplo a seguir, temos um arquivo-texto readme.txt com informações sobre a aplicação. A classe Aplicacao possui o método Sobre(), o qual lê as informações desse arquivo e as exibe na tela. Como este procedimento não envolve nenhuma propriedade de objeto, o método pode ser declarado como estático por meio do operador static. Um método estático pode acessar ainda constantes de classe e propriedades estáticas da mesma classe (por meio do operador self) e da superclasse (por meio do operador parent).

readme.txt

```
Esta aplicação está licenciada sob a GPL
Contate o autor através do e-mail autor@aplicacao.com.br
```

metodos_estaticos.php

```
<?php
class Aplicacao
{
	/* método Estático
	 * lê o arquivo readme.txt
	 */

	static function Sobre()
	{
		$fd = fopen('readme.txt', 'r') ;
		while ($linha = fgets($fd, 200))
		{
			echo $linha;
		}
	}
}

echo "Informações sobre a aplicação\n";
echo "============================\n";
Aplicacao::Sobre();

?>
```

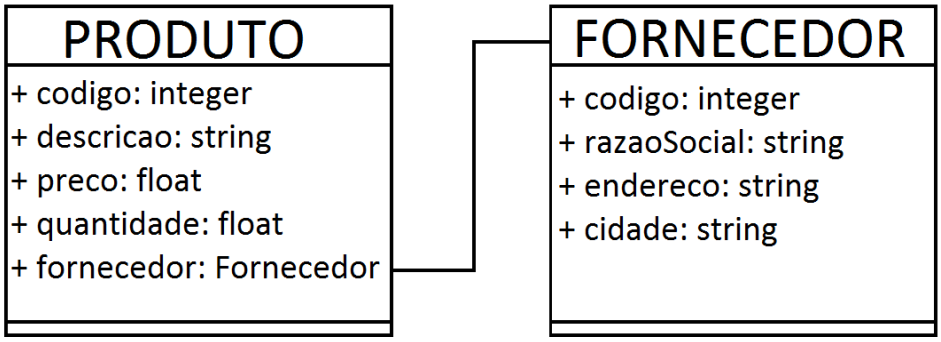
Resultado:

```
Informações sobre a aplicação
============================
Esta aplicação está licenciada sob a GPL
Contate o autor através do e-mail autor@aplicacao.com.br
```

Associação, agregação e composição

Associação

Associação é a relação mais comum entre dois objetos, de modo que um possui uma referência ã posição da memória onde o outro se encontra, podendo visualizar seus atributos ou mesmo acionar uma de suas funcionalidades (métodos). A forma mais comum de implementar uma associação é ter um objeto como atributo de outro. Veja o exemplo a seguir, no qual criamos um objeto do tipo Produto (já criado anteriormente) e outro do tipo Fornecedor. Um dos atributos do produto é o fornecedor. Observe, na imagem abaixo, como se dá a interação.



fomecedor.php

```
<?php
class Fornecedor
{
        var $codigo;
        var $razaoSocial;
        var $endereco;
        var $cidade;
}
?>
```

associacao.php

```
<?php
        include_once 'fornecedor.php';
        include_once 'produto.php';
```

```
        // instancia Fornecedor
        $fornecedor = new Fornecedor;
        $fornecedor->codigo = 222;
        $fornecedor->razaoSocial= 'Alimentos S.A.';
        $fornecedor->endereco = 'Rua Julio de Castilhos';
        $fornecedor->cidade = 'Caxias do Sul';

        // instancia Produto
        $produto = new Produto;
        $produto->codigo = 111;
        $produto->descricao = 'Doce de Uva';
        $produto->preco = 30.55;
        $produto->fornecedor = $fornecedor;

        // imprime atributos
        echo 'Código         :' . $produto->codigo . "\n";
        echo 'Descrição              :' . $produto->descricao . "\n";
        echo 'Código         :' . $produto->fornecedor->codigo . "\n";
        echo 'Razão Social :' . $produto->fornecedor->razaoSocial . "\n";
?>
```

Resultado:

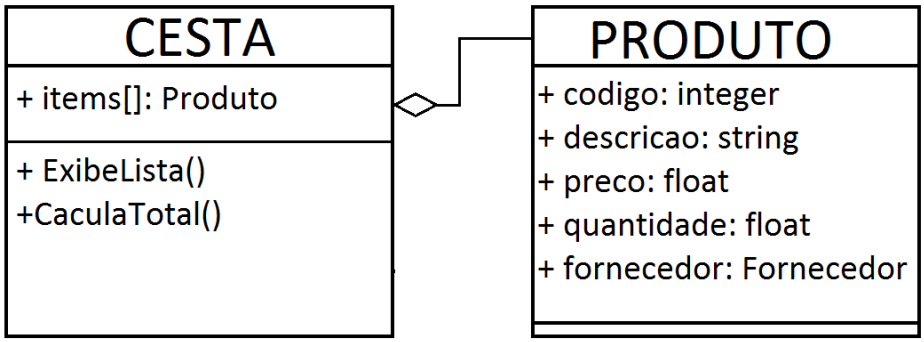
```
Código              : 111
Descrição    : Doce de Uva
Código              : 222
Razão Social: Alimentos S.A.
```

Agregação

Agregação é o tipo de relação entre objetos conhecida como todo/parte. Na agregação, um objeto agrega outro objeto, ou seja, torna um objeto externo parte de si mesmo pela utilização de um dos seus métodos. Assim, o objeto-pai poderá utilizar funcionalidades do objeto agregado. Nesta relação, um objeto poderá agregar uma ou muitas instâncias de um outro objeto. Para agregar muitas instâncias, a forma mais simples é utilizando arrays. Criamos um array como atributo da classe, sendo que o papel deste array é armazenar inúmeras instâncias de uma outra classe.

Para exemplificar esta relação utilizaremos o clássico exemplo de uma cesta de compras. A classe Cesta é o nosso todo e a classe Produto representa cada uma das partes. Uma Cesta é composta de inúmeros produtos (instâncias da classe Produto).

Para agregar objetos do tipo Produto à Cesta, criamos o método AdicionaItem() na classe Cesta, que conta também com o método ExibeLista(), o qual, na verdade, chama o método ImprimeEtiqueta() de cada um dos produtos da Cesta e o método CalculaTotal (), que soma o preço de cada um dos produtos de uma Cesta. Na imagem abaixo, temos o relacionamento entre as classes.



cesta.php

```
<?php
class Cesta
{
	private $itens;

	//adiciona itens na cesta
	function AdicionaItem(Produto $item)
	{
		$this->itens[] = $item;
	}

	//exibe a lista de produtos
	function ExibeLista()
	{
		foreach ($this->itens as $item)
		{
			$item->ImprimeEtiqueta();
		}
	}
```

```
        //calcula o valor total da cesta
        function CalculaTotal()
        {
                foreach ($this->itens as $item)
                {
                        $total += $item->preco;
                }

                return 'R$ ' . $total;
        }
}
?>
```

agregacao.php

```
<?php
include_once 'cesta.php';
include_once 'produto.php';

$produtol = new Produto;
$produtol->codigo = 1;
$produtol->descricao = 'Ameixa';
$produtol->preco = 1.40;

$produto2 = new Produto;
$produto2->codigo = 2;
$produto2->descricao = 'Morango';
$produto2->preco = 2.24;

$produto3 = new Produto;
$produto3->codigo = 3;
Sproduto3->descricao = 'Abacaxi';
$produto3->preco = 2.86;

$produto4 = new Produto;
$produto4->codigo = 4;
$produto4->descricao = 'Laranja';
$produto4->preco = 1.14;
```

```
$cesta = new Cesta;

$cesta->AdicionaItem($produtol);
$cesta->AdicionaItem($produto2);
$cesta->AdicionaItem($produto3);
$cesta->AdicionaItem($produto4);

echo $cesta->CalculaTotal() , "\n";
echo $cesta->ExibeLista();
?>
```

Resultado:

```
R$ 7,64
Código: 1
Descrição: Ameixa
Código: 2
Descrição: Morango
Código: 3
Descrição: Abacaxi
Código: 4
Descrição: Laranja
```

Talvez você já tenha percebido um possível problema neste tipo de abordagem. O que aconteceria se adicionássemos objetos que não são da classe Produto a uma Cesta? Se chamássemos métodos como CalculaTotal() e ExibeLista(), por exemplo, teríamos problemas, uma vez que eles confiam na existência do atributo Preço e do método ImprimeEtiqueta() da classe produto.

O PHP possui o conceito de TypeHinting, ou seja, "sugestão de tipo". Nesse exemplo da classe cesta, o método AdicionaItem() recebe um produto chamado $item. Veja que na frente do parâmetro indicamos a classe à qual ele deve pertencer. Caso o método AdicionaItem() seja executado passando um parâmetro que não é do tipo Produto, um erro fatal será lançado. Veja no exemplo a seguir:

Alteração na classe Cesta

```
function AdicionaItem(Produto $item)
```

agregacao2.php

```
<?php
include_once 'cesta.php';
include_once 'fornecedor.php';
```

```
include_once 'produto.php';

$fornecedor = new Fornecedor;
$fornecedor->razaoSocial = 'Produtos S.A.';

Scesta = new Cesta;
$cesta->AdicionaItem($fornecedor);
$cesta->CalculaTotal();
?>
```

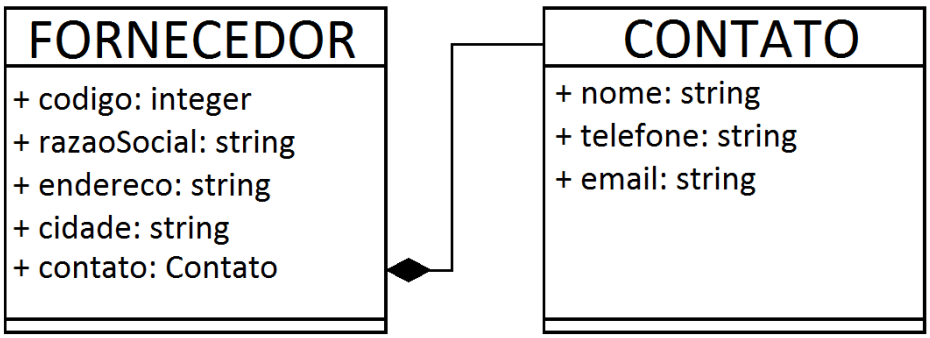
Resultado:

```
Fatal error: Argument 1 must be an instance of Produto in cesta.php on line 10
```

Composição

Composição também é uma relação que demonstra uma relação todo/parte. A diferença em relação à agregação é que, na composição, o objeto-pai ou "todo" é responsável pela criação e destruição de suas partes. O objeto-pai realmente "possui" a(s) instância(s) de suas partes. Diferentemente da agregação, na qual as instâncias do "todo" e das "partes" são independentes.

Na agregação, ao destruirmos o objeto "todo", as "partes" permanecem na memória por terem sido criadas fora do escopo da classe "todo". Já na composição, quando o objeto "todo" é destruído, suas "partes" também são, justamente por terem sido criadas pelo objeto "todo". Veja no diagrama a seguir uma composição.



contato.php

```
<?php
class Contato
{
        var $nome;
```

```
        var $telefone;
        var $email;

        //grava informações de contato
        function SetContato($nome, $telefone, $email)
        {
                $this->nome = $nome;
                $this->telefone = $telefone;
                $this->email = $email;
        }

        //obtém informações de contato
        function GetContato()
        {
        return "Nome: {$this->nome}, Telefone: {$this->telefone}, Email: {$this->email}";
        }
}
?>
```

fomecedor.php

```
<?php
class Fornecedor
{
        var $codigo;
        var $razaoSocial;
        var $endereco;
        var $cidade;
        var $contato;

        //método construtor
        function __construct(){
                //instancia novo Contato
                $this->contato = new Contato;
        }
```

```
        //grava contato
        function SetContato($nome, $telefone, $email)
        {
                //delega chamada de método
                $this->Contato->SetContato($nome, $telefone, $email);
        }

        //retorna contato
        function GetContato()
                //delega chamada de método
                return $this->contato->GetContato();
        }
}
?>
```

Neste exemplo, demonstramos a utilização de composição. Instanciamos um objeto da classe Fornecedor. A classe Fornecedor é composta de um contato, instanciado automaticamente no método construtor do Fornecedor. Veja que a classe Fornecedor é responsável por esta instância da classe Contato. Veja também que a chamada de métodos na classe principal é delegada à classe Contato.

composicao.php

```
<?php
include_once 'fornecedor.php';
include_once 'contato.php';

//instancia novo fornecedor
$fornecedor = new Fornecedor;
$fornecedor->RazaoSocial = 'Produtos S.A.';

//atribui informações de contato
$fornecedor->SetContato('Vendedor', '51 1234-5678',
'vendedor@produtos.com.br');

//imprime informações
echo $fornecedor->razaoSocial . "\n";
echo "Informações de Contato\n";
echo $fornecedor->GetContato();
?>
```

Resultado:
Produtos S.A. Informações de Contato Nome: Vendedor, Telefone: 51 1234-5678, Email: vendedor@produtos.com.br

Intercepções

O PHP implementa o conceito de interceptação em operações realizadas por objetos por meio dos métodos __set (), __get() e __call(), vistos a seguir.

Método __set()

O método __set() intercepta a atribuição de valores a propriedades do objeto. Sempre que for atribuído um valor a uma propriedade do objeto, automaticamente está atribuição passa pelo método __set(), o qual recebe o nome da propriedade e o valor a ser atribuído, podendo atribuí-lo ou não.

cachorro.php

```
<?php
class Cachorro
{
	private $nascimento;

	//método construtor
	function __construct($nome)
	{
		$this->nome = $nome;
	}

	//intercepta atribuição
	function __set($propriedade, $valor)
	{
		if($propriedade == 'nascimento')
		{
			//verifica se valor é dividido em
```

```
                        //3 partes separadas por '/'
                        if(count(explode('/', $valor))==3)
                        {
                                echo "Dado '$valor', atribuído à '$propriedade'\n";
                                $this->$propriedade = $valor;
                        }
                        else
                        {
                                echo "Dado 'Svalor', não atribuido à
'$propriedade'\n";
                        }
                }
                else
                {
                        $this->$propriedade = $valor;
                }
        }
}
?>
```

No próximo exemplo, instanciaremos um objeto $toto da classe Cachorro. A data de Nascimento do objeto será atribuída duas vezes. Note que, nas duas vezes, o método __set() irá interceptar a atribuição, validando-a. Somente a segunda atribuição passará pelas regras de atribuição definidas dentro do método __set(). A segunda data é válida porque é formada por três partes separadas por "/".

set.php

```
<?php
//inclui classe cachorro
include_once 'cachorro.php';

Stoto = new Cachorro('Totô');

$toto->nascimento = '3 de março'; // atribuição inválida
$toto->nascimento = '10/04/2005'; // atribuição correta

?>
```

Resultado:

```
Dado '3 de março', não atribuido à 'Nascimento'
```

```
Dado '10/04/2005', atribuido à 'Nascimento'
```

Método __get()

O método __get() intercepta requisições de propriedades do objeto. Sempre que for requisitada uma propriedade, automaticamente essa requisição passará pelo método __get(), o qual recebe o nome da propriedade requisitada, podendo retorná-la ou não.

produto.php

```
<?php
class Produto
{
	var $codigo;
	var $descricao;
	var $quantidade;
	private $preco;
	const MARGEM = 10;

	//método construtor de um Produto
	function __construct($codigo, $descricao, $quantidade, $preco)
	{
		$this->codigo = $codigo;
		$this->descricao = $descricao;
		$this->quantidade= $quantidade;
		$this->preco = $preco;
	}

	//intercepta a obtenção de propriedades
	function __get($propriedade)
	{
		echo "Obtendo o valor de 'Spropriedade' :\n";
		if ($propriedade == 'preco')
		{
			return Sthis->$propriedade * (1 + (self::MARGEM / 100));
		}
	}
}
```

```
?>
```

O método __get() pode ser utilizado para retornar valores calculados. Neste caso, por exemplo, ao solicitar o preço de um produto, podemos retorná-lo já com seu valor ajustado (com a margem de lucro).

get.php

```
<?php
//inclui classe Produto
include_once 'produto.php';

//cria novo produto com o preço R$ 345.67
$produto = new Produto(1, 'Pendrive 512Mb', 1, 345.67);

//imprime o preço
echo $produto->preco;
?>
```

Resultado:

```
Obtendo o valor de 'Preço' :
380,237
```

Método __call()

O método __call() intercepta a chamada a métodos. Sempre que for executado um método que não existir no objeto, automaticamente a execução será direcionada para ele, que recebe dois parâmetros, o nome do método requisitado e o parâmetro recebido, podendo decidir o procedimento a realizar.

produto.php

```
<?php
class Produto
{
        var $codigo;
        var $descricao;
        var $quantidade;
        private $preco;
        const MARGEM = 10;
```

```
        //método construtor de um Produto
        function __construct($codigo, $descricao, $quantidade, $preco)
        {
                $this->codigo = $codigo;
                $this->descricao = $descricao;
                $this->quantidade= $quantidade;
                $this->preco = $preco;
        }

        //intercepta a chamada à métodos
        function __call($metodo, $parametros)
        {
                echo "Você executou o método: {$metodo}\n";
                foreach ($parametros as $key => $parametro)
                {
                        echo "\tParâmetro $key: $parametro\n";
                }
        }
}
?>
```

Neste exemplo, iremos instanciar um objeto da classe Produto. Tentaremos executar o método Vender(), sendo que este método não existe. Note que esta execução é interceptada pelo método __call(), que emite uma mensagem na tela, informando qual foi o método executado e quais parâmetros foram passados.

call.php

```
<?php
//inclui classe Produto
include_once 'produto.php';

//criando novo produto com o preço R$ 345.67
$produto = new Produto(l, 'Pendrive 512Mb', 1, 345.67);

//executando método Vender, passando 10 unidades.
echo $produto->Vender(10);
```

```
?>
```

Resultado:

```
Você executou o método: Vender
Parâmetro 0: 10
```

Método __toString()

Quando imprimimos objetos na tela, por meio de comandos como o echo e print, o PHP exibe no console o identificador interno do objeto, como nos exemplos a seguir:

```
Object id #1
Object id #2
```

Para alterar esse comportamento, podemos definir o método __tostring() para cada classe. Caso o método __tostring() exista, no momento em que mandarmos exibir um objeto no console, o PHP irá imprimir o retorno dessa função.

No exemplo a seguir, o método __tostring() retorna a propriedade $nome, que é definida no método construtor do objeto. Nesses casos, instanciaremos dois objetos: $toto e $vava. Note que no momento em que imprimimos esses objetos no console, será retornado o próprio nome dos mesmos.

tostring.php

```
<?php
class Cachorro
{
        private $nascimento;

        //método construtor
        function __construct($nome)
        {
                $this->nome = $nome;
        }

        //tostring, executado sempre que o objeto for impresso
        function __tostring()
        {
                return $this->nome;
        }
```

```
}

$toto = new Cachorro('Totó');
$vava = new Cachorro('Vava');

echo $toto;
echo "\n";
echo $vava;
echo "\n";
?>
```

Resultado:

```
Totó
Vava
```

Método __clone()

O comportamento-padrão do PHP quando atribuímos um objeto ao outro é criar uma referência entre os objetos. Dessa forma, teremos duas variáveis apontando para a mesma região da memória. Este é o comportamento desejado na maioria das vezes, mas como proceder quando quisermos de fato duplicar um objeto na memória? Simples! Utilizaremos o operador clone. O objeto resultante será idêntico ao original, a menos que tenhamos declarado o método __clone(), responsável por definir o comportamento da ação de clonagem, atuando diretamente nas propriedades do objeto resultante dessa ação.

No exemplo, criaremos um objeto $toto da classe Cachorro e, em seguida, iremos cloná-lo. Neste exemplo, o método clone() executado no momento da clonagem define que o código da coleira do Cachorro deverá ser incrementado, sua idade será zerada e seu nome será acrescido de 'Júnior' ao final.

clone.php

```
<?
class Cachorro
{

        //método construtor
        function __construct($coleira, $nome, $idade, $raca)
        {
                $this->coleira = $coleira;
```

```
                $this->nome = $nome;
                $this->idade = $idade;
                $this->raca = $raca;
        }

        function __clone()
        {
                $this->coleira = $this->coleira +1;
                $this->nome .= ' Junior';
                $this->idade = 0;
        }
}

$toto = new Cachorro(100, 'Totó', 10, 'Fox Terrier');
$vava = clone $toto;

echo 'Código: ' . $toto->coleira . "\n";
echo 'Nome: ' . $toto->nome . "\n";
echo 'Idade: ' . $toto->idade . " anos \n";
echo "\n";
echo 'Código: ' . $vava->coleira . "\n";
echo 'Nome: ' . $vava->nome . "\n";
echo 'Idade: ' . $vava->idade . " anos \n";
?>
```

Resultado:

```
Código: 100
Nome: Totó
Idade: 10 anos

Código: 101
Nome: Totó Junior
Idade: 0 anos
```

Interfaces

A programação orientada a objetos baseia-se fortemente na interação de classes e objetos. Um objeto deve conhecer quais são as funcionalidades que um outro objeto pode lhe fornecer (métodos). Na etapa de projeto do sistema, podemos definir conjuntos de métodos que determinadas classes do nosso sistema deverão implementar incondicionalmente. Tais conjuntos de métodos são as interfaces, as quais contêm a declaração de métodos de forma prototipada, sem qualquer implementação. Toda classe que implementar uma interface deverá obrigatoriamente possuir os métodos predefinidos na interface; caso contrário, resultará em erro.

No exemplo a seguir, criaremos a classe Aluno, mas o projeto do sistema indica que esta classe deve implementar um conjunto de métodos que devem estar disponíveis para os outros objetos do sistema. Para tanto, criamos a interface IAluno contendo os métodos GetNome(), SetNome() e SetResponsavel(). Neste exemplo, a classe Aluno não está implementando o método SetResponsavel () (que recebe um parâmetro da classe Pessoa), dessa forma, o programa irá retornar o erro indicado no final.

Observação: uma classe pode implementar diversas interfaces separadas por vírgula.

Ialuno.php

```
<?php
interface IAluno
{
        function SetNome();
        function SetNome($nome);
        function SetResponsavel(Pessoa Sresponsavel);
}
?>
```

interface.php

```
<?php
//inclui a interface IAluno
include_once 'ialuno.php';

//Classe Aluno
class Aluno implements IAluno
{
        //atribui o nome do aluno
        function SetNome($nome)
```

```
        {
                $this->nome = $nome;
        }

        //retorna o nome do aluno
        function SetNome()
        {
                return $this->nome;
        }
}

//instancia novo Aluno
$aluno = new Aluno;

//chama métodos quaisquer
$aluno->setNome('Joana');
echo $aluno->getNome();

?>
```

Resultado:

```
Fatal error: Class Aluno contains 1 abstract methods and must therefore be declared abstract (IAluno::SetResponslavel) in interface.php on line 19
```

Objetos dinâmicos

O PHP nos oferece diversas facilidades para manipulação de objetos. Uma delas é a possibilidade de criar objetos dinamicamente, sem ter a classe previamente definida. Naturalmente, esses objetos irão armazenar somente dados, e não funções.

Neste primeiro exemplo, criaremos dois objetos, $william e $rafaela, com seus respectivos atributos. Note que esses objetos são da classe stdClass (Standard Class), classe vazia criada especialmente para esses propósitos.

Objetos_dinamicos.php

```
<?php

//cria objeto william
$william->nome = 'William Santos';
```

```
$william->idade = 20;
$william->profissao = 'Programador';

//cria objeto rafaela
$rafaela->nome = 'Rafaela Santos';
$rafaela->idade = 24;
$rafaela->profissao = 'Policial Militar';

print_r($william);
print_r($rafaela);
?>
```

Resultado:

```
stdClass Object
(
        [nome] => William Santos
        [idade] => 20
        [profissão] => Programador
)
stdClass Object
(
        [nome] => Rafaela Santos
        [idade] => 24
        [profissão] => Policial Militar
)
```

Outra possibilidade que o PHP nos oferece é utilizar variáveis variantes para declarar as propriedades de um objeto.

objarray.php

```
<?php
        // cria array dados_william
        $dados_william['nome'] = 'William Santos';
        $dados_wi11iam['idade'] = 20;
        $dados_william['profissão'] = 'Programador';

        // cria array dados_rafaela
        $dados_rafaela['nome'] = 'Rafaela Santos';
```

```
        $dados_rafaela['idade'] = 24;
        $dados_rafaela['profissão'] = 'Policial Militar';

        // cria objeto william
        foreach ($dados_william as Schave => $valor)
        {
                // utiliza variáveis variantes
                $william->$chave = $valor;
        }

        // cria objeto rafaela
        foreach ($dados_rafaela as $chave => Svalor)
        {
                // utiliza variáveis variantes
                $rafaela->$chave = $valor;
        }

        echo "{Swilliam->nome} é {$william->profissao}\n";
        echo "{$rafaela->nome} é {$rafaela->profissao}\n";
?>
```

Resultado:

```
William Santos é Programador
Rafaela Santos é Policial Militar
```

Tratamento de erros

Existem diversas formas de realizar tratamento de erros em PHP. Veremos as formas mais comuns utilizadas, finalizando com o tratamento de exceções.

A função die()

A forma de manipulação de erro mais simples é abortar a execução da aplicação. Claro que não é qualquer erro que deva causar esta interrupção. Esta é a forma mais simplista de tratar erros e, portanto, não deve ser utilizada amplamente.

funcao_die.php

```
<?php
function Abrir($file = null)
{
	if(!$file)
		die('Falta o parâmetro com o nome do Arquivo');

	if(!file_exists($file))
		die('Arquivo não existente');

	if(!$retorno = @file_get_contents($file))
		die('Impossivel ler o arquivo');

	return $retorno;
}

//abrindo um arquivo
$arquivo = Abrir('/tmp/arquivo.dat');
echo $arquivo;
?>
```

Resultado:

```
Arquivo não existente
```

No exemplo anterior vimos que, como o referido arquivo não existe, a aplicação seria abortada com a mensagem de erro. Caso faltasse o parâmetro indicando o nome do arquivo, a aplicação seria abortada com a mensagem "Falta o parâmetro com o nome do Arquivo" e, caso não fosse possível efetuar a leitura do mesmo por problemas como falta de direitos, a aplicação seria abortada com a mensagem "Impossível ler o arquivo".

Utilizar a função die() para controlar erros é ruim porque ela simplesmente aborta a execução do programa, o que, na maioria dos casos, não é o comportamento desejado, visto que nem todos os tipos de erros são fatais para a execução da aplicação.

Retorno deflags

A segunda forma de manipulação de erros é o retorno de flags TRUE ou FALSE. Retornamos TRUE em caso de sucesso na operação e FALSE em caso de erros.

retorno_flag.php

```
<?php
function Abrir($file = null)
{
	if(!$file)
	{
		return false;
	}

	if(!file_exists($file))
	{
		return false;
	}

	if(!$retorno = @file_get_contents($file))
	{
		return false;
	}

	return $retorno;
}

$arquivo = Abrir('/tmp/arquivo.dat');

// verificando se abriu o arquivo.
if(!$arquivo)
{
	echo 'Falha ao abrir o arquivo';
}
else
{
	echo $arquivo;
}
?>
```

Resultado:

```
Falha ao abrir o arquivo
```

A vantagem desse tipo de abordagem em relação ao primeiro é que a aplicação segue a sua execução sem ser abortada (a não ser que a abortemos ao testar o retorno da função). A desvantagem é que não sabemos exatamente em qual ponto do programa a execução falhou, não tendo como exibir a mensagem de erro correta, apenas uma genérica.

Lançando erros

Uma forma mais elegante de realizar a manipulação de erros é por meio das funções trigger_error(), que lança um erro, e a função set_error_handler(), a qual define uma função que realizará a manipulação dos erros lançados.

lancando_erro.php

```
<?php
function Abrir($file = null)
{
	if(!$file)
	{
		trigger_error('Falta o parâmetro com o nome do Arquivo', E_USER_NOTICE);
		return false;
	}

	if(!file_exists($file))
	{
		trigger_error('Arquivo não existente', E_USER_ERROR);
		return false;
	}

	if(!$retorno = @file_get_contents($file))
	{
		trigger_error('Impossivel ler o arquivo', E_USER_WARNING);
		return false;
	}

	return $retorno;
}

// função para manipular o erro
```

```
function manipula_erro($numero, $mensagem, $arquivo, $linha)
{
	$mensagem = "Arquivo $arquivo : linha $linha # no. $numero : $mensagem\n";

	// escreve no log todo tipo de erro
	$log = fopen('erros.log', 'a');
	fwrite($log, $mensagem);
	fclose($log);

	// se for uma warning
	if($numero == E_USER_WARNING)
	{
		echo $mensagem;
	}

	// se for um erro fatal
	else if (ínumero == E_USER_ERROR)
	{
		echo $mensagem;
		die;
	}
}

// define a função manipula_erro como manipuladora dos erros ocorridos
set_error_handler('manipula_erro');

// abrindo um arquivo
$arquivo = Abrir('/tmp/arquivo.dat');
echo $arquivo;

?>
```

Resultado:

```
Arquivo lancando_erro.php : linha 11 # no. 256 : Arquivo não existente
```

A vantagem desse tipo de abordagem para manipulação de erros é a liberdade que temos para personalizar o tratamento de erros por meio da função manipula_erro() definida por set_error_handler() como sendo a função a ser invocada quando algum erro ocorrer. Dentro

desta função (que recebe o arquivo e a linha em que ocorreu o erro, além do número e a mensagem de erro ocorrido) podemos exibir ou suprimir a exibição do erro, gravá-lo em um banco de dados ou gravar em um arquivo de log, como fizemos no exemplo demonstrado. No exemplo, todas as mensagens (ERROR, WARNING e NOTICE) são armazenadas em um arquivo de log; somente as de WARNING e ERROR são exibidas na tela, e as de ERROR causam parada na execução da aplicação pelo comando die. A desvantagem deste tipo de abordagem é que concentramos todo tratamento de erro em uma única função genérica, quando muitas vezes precisamos analisar caso a caso para optar por uma determinada ação.

Tratamento de exceções

O PHP implementa o conceito de tratamento de exceções, da mesma forma que ele é implementado em linguagens como C++ ou Java. Uma exceção é um objeto especial derivado da classe Exception, que contém alguns métodos para informar ao programador um relato do que aconteceu. A seguir, você confere estes métodos:

Método	Descrição
getMessage()	Retorna a mensagem de erro.
getCode()	Retorna o código de erro.
getFile()	Retorna o arquivo no qual ocorreu o erro.
getLine()	Retorna a linha na qual ocorreu o erro.
GetTrace()	Retorna um array com as ações até o erro.
getTraceAsString()	Retorna as ações em forma de string.

O tratamento de erros ocorre em dois níveis. Quando executamos um conjunto de operações que pode resultar em algum tipo de erro, monitoramos essas operações, escrevendo o código dentro do bloco try. Dentro das operações críticas, quando ocorrer algum erro, devemos fazer uso do comando throw para "lançar" uma exceção (objeto Exception), isto é, para interromper a execução do bloco contido na cláusula try, a qual recebe esta exceção e repassa para outro bloco de comandos catch. Dentro do bloco de comandos catch, programamos o que deve ser realizado quando da ocorrência da exceção, podendo emitir uma mensagem ao usuário, interromper a execução da aplicação, escrever um arquivo de log no disco, dentre outros.

O interessante desta abordagem é que a ação resultante do erro ocorrido fica totalmente isolada, externa ao contexto do código gerador da exceção. Esta modularidade permite mudarmos o comportamento de como é realizado este tratamento de erros, sem alterar o bloco código principal.

exception.php

```
<?php
function Abrir($file = null)
{
	if(!$file)
	{
		throw new Exception('Falta o parâmetro com o nome do arquivo');
	}
	if(!file_exists($file))
	{
		throw new Exception('Arquivo não existente');
	}
	if(!$retorno = @file_get_contents($file))
	{
		throw new Exception('Impossível ler o arquivo');
	}
	return $retorno;
}

// abrindo um arquivo com tratamento de exceções
try
{
	$arquivo = Abrir('/tmp/arquivo.dat');
	echo $arquivo;
}
// captura o erro
catch (Exception $exceção)
{
	echo $exceção->getFile() . ' : ' . $exceção->getLine() . ' # ' . $exceção->getMessage();
}
?>
```

Resultado:

```
exception.php : 10 # Arquivo não existente
```

No exemplo anterior, capturamos o erro no bloco try{} e catch{}, mas ainda não temos uma boa separação para cada tipo de erro ocorrido. Para realizar tal separação, podemos lançar tipos customizados para cada tipo de erro. No exemplo a seguir, criaremos três tipos de erros a serem lançados ParameterException, FileNotFoundException e

FilePermissionException. Para tanto, iremos declarar essas classes por meio do mecanismo de herança, a partir da superclasse Exception. Quando uma exceção do tipo ParameterException for lançada, não ocorrerá nada. Quando uma exceção do tipo FileNotFoundException for lançada, todas as informações a respeito do erro serão exibidas, a aplicação será terminada e, quando uma exceção do tipo FilePermissionException for lançada, somente a mensagem de erro será exibida.

subexception.php

```
<?php
function Abrir($file = null)
{
	if(!$file)
	{
		throw new ParameterException('Falta o parâmetro com o nome do
arquivo');
	}
	if(!file_exists($file))
	{
		throw new FileNotFoundException('Arquivo não existente');
	}
	if(!$retorno = @file_get_contents($file))
	{
		throw new FilePermissionException('Impossivel ler o arquivo');
	}
	return Sretorno;
}



// definição das subclasses de erro
class ParameterException extends Exception!}
class FileNotFoundException extends Exception!}
class FilePermissionException extends Exception!}



// abrindo um arquivo com tratamento de exceções
try
{
```

```
        $arquivo = Abrir('/tmp/arquivo.dat');
        echo $arquivo;
}
// captura o erro
catch (ParameterException $exceção)
{
        // não faz nada...
}
catch (FileNotFoundException $exceção)
{
        var_dump($exceção->getTrace());
        echo "finalizando aplicação...\n";
        die;
}
catch (FilePermissionException $exceção)
{
        echo $exceção->getFile() . ' : ' . $exceção->getLine() . ' # ' . $exceção->getMessage();
}
?>
```

Resultado:

```
array(1) {
        [0] => array(4) {
                ["file"]=> string(16) "subexception.php"
                ["line"]=> int(28)
                ["function"]=> string(5) "Abrir"
                ["args"]=> array(1) {
                        [0]=> string(16) "/tmp/arquivo.dat"
                }
        }
}
finalizando aplicação...
```

PDO

Introdução

O PHP é, em sua maioria, um projeto voluntário cujos colaboradores estão distribuídos geograficamente ao redor de todo o planeta. Como resultado, o PHP evoluiu baseado em necessidades individuais para resolver problemas pontuais, movidos por razões diversas. Por um lado, essas colaborações fizeram o PHP crescer rapidamente;

por outro, geraram uma fragmentação das extensões de acesso à base de dados, cada qual com sua implementação particular, não havendo real consistência entre as interfaces das mesmas (Oracle, MySQL, PostgreSQL, SQL Server etc.). Em razão da crescente adoção do PHP, surgiu a necessidade de unificar o acesso às diferentes extensões de bancos de dados presentes no PHP.

Assim surgiu a PDO (PHP Data Objects), cujo objetivo é prover uma API limpa e consistente, unificando a maioria das características presentes nas extensões de acesso a banco de dados.

A PDO não é uma biblioteca completa para abstração do acesso à base de dados, uma vez que ela não faz a leitura e tradução das instruções SQL, adaptando-as aos mais diversos drivers de bancos de dados existentes. Ela simplesmente unifica a chamada de métodos, delegando-os para as suas extensões correspondentes e faz uso do que há de mais recente no que diz respeito à orientação a objetos presente no PHP.

Para conectar em bancos de dados diferentes, a única mudança é na string de conexão. Veja a seguir alguns exemplos nos mais diversos bancos de dados.

Banco	String de Conexão
SQLite	new PDO('sqlite:teste.db');
FireBird	new PDO("firebird:dbname=C:\\base.GDB", "SYSDBA", "masterkey");
MySQL	new PDO('mysql:host=localhost;port=3306;dbname=livro', 'user', 'senha');
Postgres	new PDO(‘pgsql:dbname=example;user=user;password=senha;host=localhost');

Para podermos utilizar o PDO, deve-se ter habilitado no php.ini as bibliotecas (drivers) de acesso para o banco de dados de nossa aplicação. No Linux, habilitamos da seguinte forma:

```
extension=mysql.so
extension=pgsql.so
extension=sqlite.so
extensi on=pdo_mysql.so
extension=pdo_pgsql.so
extensi on=pdo_sqli te.so
```

No Windows, entretanto, habilitamos da seguinte forma:

```
extension=php_mysql.dl 1
extension=php_pgsql.dll
extensi on=php_sqli te.dll
extension=php_pdo_mysql.dll
extension=php_pdo_pgsql.dll
extensi on=php_pdo_sqli te.dl 1
```

Exemplos

Para facilitar a aprendizagem da PDO, veremos uma série de exemplos, cada qual abordando um aspecto diferente relacionado a bancos de dados. Por ora veremos como se dá uma inserção e, posteriormente, veremos uma listagem de dados.

Inserção, alteração e exclusão

Neste primeiro exemplo, assim como no exemplo de inserção de dados visto anteriormente, o programa irá se conectar ao banco de dados em local host, chamado livro, com o usuário postgres. Em seguida, ele irá inserir dados relativos a clientes na base de dados. Por fim, ele fechará a conexão ao banco de dados. Caso ocorra algum erro durante a execução das instruções SQL, será gerada uma exceção, tratada adequadamente pelo bloco try/catch.

pdo_insere.php

```
<?php
try
{
        // instancia objeto PDO, conectando no postgresql
        $conn                                   =                                   new
PDO('pgsql:dbname=livro;user=postgres;password=;host=localhost');

        // executa uma série de instruções SQL
```

```
        $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (1, 'Érico Veríssimo')");
        $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (2, 'John Lennon1)");
        $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (3, 'Mahatma Gandhi')");
        $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (4, 'Ayrton Senna')");
        $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (5, 'Charlie Chaplin')");
        $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (6, 'Anita Garibaldi')");
        $conn->exec("INSERT INTO famosos (codigo, nome) VALUES (7, 'Mário Quintana')");

        // fecha a conexão
        $conn = null ;
}
catch (PDOException $e)
{
        // caso ocorra uma exceção, exibe na tela
        print "Erro!: " . $e->getMessage() . "\n";
        die();
}
?>
```

Listagens

No próximo exemplo, assim como no programa de listagem de dados visto anteriormente, o programa se conectará ao banco de dados livro. Em seguida, ele irá selecionar código e nome dos famosos existentes nesse banco de dados e exibir na tela.

Veja como os resultados são percorridos via laço de repetições (FOREACH) por meio de uma simples iteração. Cada linha (resultset) do resultado é retornada para dentro do array $row, indexado pelos nomes das colunas do SELECT ("codigo", "nome") e também por índices numéricos que representam as posições das colunas (0,1,...). Novamente o controle de erros é realizado por tratamento de exceções. Caso alguma exceção seja gerada, a sua mensagem será exibida na tela pelo bloco catch.

pdo_lista.php

```
<?php
```

```
try
{
	// instancia objeto PDO, conectando no Postgresql
	$conn = new PD0('pgsql:dbname=livro;user=postgres;password=;host=localhost');

	// executa uma instrução SQL de consulta
	$result = $conn->query("SELECT codigo, nome FROM famosos");
	if($result)
	{
		// percorre os resultados via iteração
		foreach($result as Srow)
		{
			// exibe os resultados
			echo $row['codigo'] . ' - ' . $row['nome'] . "<br>\n";
		}
	}
	// fecha a conexão
	$conn = null;
}
catch (PDOException $e)
{
	print "Erro!: " . $e->getMessage() . "<br/>";
	die();
}
?>
```

Resultado:

```
1 - Érico Veríssimo
2 - lohn Lennon
3 - Mahatma Gandhi
4 - Ayrton Senna
5 - Charlie Chaplin
6 - Anita Garibaldi
7 - Mário Quintana
```

Podemos também retornar os dados de uma consulta por meio da função fetch(), a qual aceita como parâmetro o "estilo de fetch". De acordo com o estilo, o retorno poderá ser um dos relacionados a seguir:

Parâmetros	Descrição
PDO:: FETCH_ASSOC	Retorna um array indexado pelo nome da coluna.
PDO:: FETCH_NUM	Retorna um array indexado pela posição numérica da coluna.
PDO:: FETCH_BOTH	Retorna um array indexado pelo nome da coluna e pela posição numérica da mesma.
PDO:: FETCH_0BJ	Retorna um objeto anônimo (StdClass), de modo que cada coluna é acessada como uma propriedade.

Neste exemplo a seguir, repetimos o mesmo programa que lista os códigos e os nomes dos famosos do banco de dados. A diferença é que agora utilizaremos a função fetch() para iterar os resultados. O estilo de fetch desejado será o PDO: :FETCH_OBJ, que retorna os dados da consulta em forma de objeto. Veja que agora a variável $row é um objeto, e cada coluna retornada (codigo, nome) é acessada como uma propriedade deste objeto.

pdo_lista_obj.php

```
<?php
try
{
	// instancia objeto PDO, conectando no Postgresql
	$conn = new PDO('pgsql:dbname=livro;user=postgres;password=;host=localhost');

	// executa uma instrução SQL de consulta
	$result = $conn->query("SELECT codigo, nome FROM famosos");
	if($result)
	{
		// percorre os resultados via fetch()
		while ($row = $result->fetch(PDO::FETCH_0BJ))
		{
			// exibe os dados na tela, acessando o objeto retornado
			echo $row->codigo . ' - ' . $row->nome . "<br>\n";
		}
	}

	// fecha a conexão
	$conn = null;
```

```
}
catch (PDOException $e)
{
        print "Erro!: " . $e->getMessage() . "<br/>";
        die();
}
?>
```

Resultado:

```
1 - Érico Veríssimo
2 - John Lennon
3 - Mahatma Gandhi
4 - Ayrton Senna
5 - Charlie Chaplin
6 - Anita Garibaldi
7 - Mário Quintana
```

Reescreveremos, então, o programa de inserção de dados, que anteriormente foi escrito para PostgreSQL, agora em MySQL. Veja que a única diferença entre um programa e outro é a linha em que instanciamos o objeto PDO (new PDO), de modo que os parâmetros são diferentes. Todo o restante do programa é simplesmente igual em razão da interface clara e unificada da biblioteca PDO. Veja a seguir como fica nosso programa que insere dados na tabela de famosos adaptado para o uso com MySQL.

pdo_insere_my.php

```
<?php
try
{
        // instancia objeto PDO, conectando no mysql
        $conn = new PD0('mysql:host=localhost;port=3306;dbname=livro', 'root', 'mysql');


        // executa uma série de instruções SQL
        $conn->exec("INSERT INTO famosos (codigo,nome) VALUES (1, 'Érico Veríssimo')");
        $conn->exec("INSERT INTO famosos (codigo,nome) VALUES (2, 'John Lennon')");
        $conn->exec("INSERT INTO famosos (codigo,nome) VALUES (3, 'Mahatma Gandhi')");
        $conn->exec("INSERT INTO famosos (codigo,nome) VALUES (4, 'Ayrton Senna')");
```

```
        $conn->exec("INSERT INTO famosos (codigo,nome) VALUES (5, 'Charlie Chaplin')");
        $conn->exec("INSERT INTO famosos (codigo,nome) VALUES (6, 'Anita Garibaldi')");
        $conn->exec("INSERT INTO famosos (codigo,nome) VALUES (7, 'Mário Quintana')");

        // fecha a conexão
        $conn = null;
}
catch (PDOException $e)
{
        // caso ocorra uma exceção, exibe na tela
        print "Erro!: " . $e->getMessage() . "\n";
        die();
}
?>
```