No que diz respeito ao desempenho, ao compromisso, ao esforço, à dedicação, não existe meio termo. Ou você faz uma coisa bem-feita ou não faz. Ayrton Senna

**2 – Orientação a Objetos**

A orientação a objetos é um paradigma que representa toda uma filosofia para construção de sistemas. Em vez de construir um sistema formado por um conjunto de procedimentos e variáveis nem sempre agrupadas de acordo com o contexto, como se fazia nas linguagens estruturadas (Cobol, Clipper, Pascal), na orientação a objetos utilizamos uma ótima mais próxima no mundo real. Liamos com objetos, estrutudas que já conhecemos no nosso dia-a-dia sobre as quais possuímos maior compreensão.

**2.1 – Introdução**

**2.1.1 – Programação Estruturada**

**2.1.2 – 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.

**2.2 – 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 de objetos. Tem-se por propriedades características intrínsecas à classe em questão.

**2.3 – 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 objeto 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.

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** seguindo do nome da classe que desejamos instanciar.

**2.4 – 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 **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 crido são inicializadas com o valor **NULL**.

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 bojeto é deslocado da memória, quando atribuímos o valor NULL ao objeto, quando utilizamos a função unset() sobre o mesmo ou, em ultima instancia, quando o programa é finalizado.

Para definir um método destrutor em uma determinada classe basta declarar o método **\_\_destruct().**

**2.5 – Herança**

Utilizando a herança, em vez de criarmos uma estrutura totalmente nova (uma classe), podemos reaproveitar uma estrutura já existente que nos forneca uma base abastrata para o desenvolvimento, provendo recursos básicos e comum.

Vamos 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.

**2.6 – Polimorfismo**

Polimorfismo em orientação a objetos é o prinípio 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.

**2.7 – Abstração**

No paradigma de orientação a objetos se prega o conceito da “abstração”. De acordo com o dicionário Priberam, “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 peca monolítica; devemos separá-lo em partes, concentrando-nos nas pecas mais importantes e ignorando os detalhes (em primeiro momento), para podermos construir pecas bem-definicas que possam ser reaproveitadas mais tarde, formando uma estrutura hierárquica.

**2.7.1 – Classes abstratas**

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

**2.7.2 – 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 ser especializada.

**2.7.3 – Métodos abstratos**

Um métodos 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.

**2.7.4 – 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 inicio da sua declaração.

**2.8 – Encapsulamento**

Um dos recursos mais interessantes na orientação a objetos é o encapsulamento, um mecanismo que provê a proteção de acesso aos membros internos de um objeto. Lembre-se que uma classe possui responsabilidade sobre os atributos que contém. Desta 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 deve 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 seu 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:

|  |  |
| --- | --- |
| Visibilidade | Descrição |
| **private** | Membros declarados como **private** somente podem ser acessados dentro da própria classe em que foram declarados.  Não poderão ser acessados de classes descendentes nem a partir do programa que fez uso desta 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. 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 a propriedade. |
| **public** | Membros declarados como **public** poderão ser acessados livremente a partir da própria classe em que foram declarados, partir de classes descendentes e a partir do programa que faz uso desta classe (maquilando o objeto em si). Na UML, simbolizamos com um (+) em frente a propriedade |

**2.9 – 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.

**2.9.1 – Constantes**

No exemplo a seguir, veremos como se dá a declaração de uma constante pelo operador **cont**, 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.

**2.9.2 – 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 delas derivados, propriedades estáticas são compartilhadas entre todos os objetos de uma mesma classe.

**2.9.3 – Métodos estáticos**

Métodos estáticos podem ser invocados diretamente da classe, sem a necessidade de instanciar um objeto para isto. Eles não devem referenciar propriedades internas pelo operador **$this**, porque este operador é utilizado para referenciar instancias 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().**

**2.10. – Associação, agregação e composição**

Em Orientação a Objeto é comum que objetos se relacionem um com os outros para desempenhar suas tarefas.

**Associação**: É a relação mais comum entre dois objetos. Quando acontece? Quando uma classe “usa” outra classe.

Exemplo: *Um motorista utiliza (depende do carro, mas este não faz parte do motorista) o carro para trabalhar.*

**Agregação**: É o tipo de relação conhecida como todo/parte. Na agregação o objeto agrega outro objeto, ou seja, torna um objeto externo parte de si.. Assim o objeto pai poderá utilizar funcionalidades do objeto agregado, mas ele não depende dele (agregado) para existir.

Exemplo: *A tampa de uma caneta é parte da caneta, se a caneta não tiver a tampa, ela não deixará de ser uma caneta.*

**Composição**: Neste caso, o objeto-pai é responsável pela criação e destruição de suas partes. O objeto-pai realmente “possui” as instâncias de suas partes. Resumidamente, um só existe caso o outro também exista.

Exemplo: *O motor de um carro é parte fundamental do carro.*

**2.10.1 – Associação**

Associação é a relação ais comum entre dois objetos, de modo que um possui uma referencia à composição de memória onde o outro se encontra, podemos 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.

**2.10.2 – 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 mais instancias de um outro objeto. Para agregar muitas instancias, a forma mais simples é utilizando arrays. Criamos um array como atributo da classe, sendo que o papel deste array é armazenar instancias de uma outra classe.

**2.10.3 – Composição**

Composição também é uma relação que demostra 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. Diferente 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”

**2.11 – Intercepções**

O PHP5 introduziu o conceito de interceptação em operações por objetos por meio dos métodos \_\_set(), \_\_get() e \_\_call(). Vistos a seguir.

**2.11.1 – Método** \_\_set( )

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

**2.11.2 – Método** \_\_get()

O método \_\_set() intercepta requisições da propriedade 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.

**2.11.3 – 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.

**2.11.4 – 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.

**2.11.5 – Método** toXml()**?**

O PHP não possui um método para retornar o objeto formato XML, mas isso não é problema, uma vez que é muito fácil desenvolver tal funcionalidade.

**2.12 – Interfaces**

**2.13 – Método** \_\_clone()

O comportamento-padrão do PHP quando atribuímos um objeto a outro é criar uma referência entre os objetos. Desta 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.

**2.14 – Autoload**

Sempre que se instancia um objeto em PHP, é necessário ter a declaração da classe na memória; caso contrário, a aplicação é finalizada com um erro. Para introduzir uma classe na memória podemos utilizar comando como **include\_once**. Podemos realizar tal operação no início da aplicação introduzindo todas as classes que poderão ser necessárias durante a execução da aplicação. Outra forma seria introduzir a classe na linha imediatamente anterior à instancia do objeto.

Para simplificar tal procedimento, o PHP introduz a função \_\_autoload(), ou “carga automática”. Com ela, a carga da classe é realizada de forma dinâmica sempre que um objeto for instanciado. Essa função recebe o nome da Classe que está para ser instanciada, mas antes que isto aconteça, ela tem de introduzir sua classe na memória.

**2.15 – 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.

**2.16 – Manipulação de XML**

Nesta seção abordaremos um conjunto de funções chamadas de SimpleXML, disponíveis a partir do PHP5. Seu objetivo, como diz o seu nome, é tornar simples a tarefas de carregar, interpretar e alterar documentos XML. É de suma importância a manipulação de arquivos XML no desenvolvimento de aplicações, seja para armazenamento de arquivos de configuração, de dados reais ou mesmo para o intercâmbio de informações entre diferentes aplicações.

**2.16.1 – Exemplos**

A função simplexml\_loag\_file() realiza a leitura de um documento XML, criando um objeto do tipo SimpleXmlElement a partir desta operação. Caso o documento seja mal formatado, ou mesmo não seja um documento XML, essa função retorna FALSE.

Um objeto SimpleXMLElement representa um elemento de um documento XML, o qual, quando interpretado pelas funções simplexml\_load\_file() ou simplexml\_load\_string(), resulta em um objeto do tipo SimpleXMLElement internos, representando subelementos (nodos). A segui, estão alguns dos métodos providos por um objeto do tipo SimpleXMLElement:

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| **asXML()** | Retorna uma string XML formatada representando o objeto, bem como seus subelementos. |
| **attributtes()** | Lisa os atributos definidos dentro da tag XML. |
| **children()** | Retorna os elementos-filho do objeto (subnodos), bem como seus valores |
| **addChild(**  string *nome,*  string *valor,*  string *namesapce***)** | Adiciona um elemento ao nodo especificado e retorna um objeto do tipo SimpleXmlElement. |

**2.17 – Tratamento de erros**

Aa

**2.17.1 - A função die()**

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

**2.17.2 – Retorno de flags**

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 casos de erros.

A vantagem deste 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.

**2.17.3 – 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 defini uma função que realizará a manipulação dos erros lançados

trigger\_error() – gera um erro

set\_error\_handler() – define uma função a ser utilizada para maniupular erros.

**Traduções:**

handler = manipulador, treinador, domador

trigger = gatilho,

**2.17.4 – Tratamento de exceções**

O PHP5 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 aos 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. |
| **gerLine()** | 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 poder resultar em algum tipo de erro, monitoramos essas operações escrevendo o código dentro do bloco ***try***. Dentro das operações críticas, quando ocorrer algum erro, deveremos fazer o uso do momando ***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, dentro outros.

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

**Traduções:**

Exception = exceção

throw = jogar lançar

try = tentar fazer, fazer uma tentativa, experimentar, tentar

catch = pegar, apanhar, agarrar