Diego L. Alexandre

Diego.alexandre.work@gmail.com

Orientação a objetos

POO com C#

**Orientação a Objetos em .Net**

**O que são os paradigmas de programação**

São modos de diferentes de montara estrutura e execução de um código. Um paradigma de programação fornece e determina a visão a visão que o programador possui sobre a estruturação e execução do programa.

Cada paradigma surgiu de necessidades diferentes. Dado isso, cada um apresenta maiores vantagens sobre outros dentro do desenvolvimento de determinado sistema. Sendo assim, um paradigma pode oferecer técnicas apropriadas para uma aplicação específica. Entre eles estão a programação orientada a objetos (POO), e a programação estruturada (PE).

**Programação Estruturada (PE)**

A programação estruturada tem como principal característica sua interpretação linha por linha, em pequenos trechos de código, podendo eles não estar em uma ordem específica. Há três tipos de estruturas básicas para navegar pelo código: **sequência**, **seleção** e **repetição**.

* **Sequências:** são os comandos a serem executados de cima para baixo, linha a linha, do programa, de forma sequencial;
* **Seleções:** sequências que só devem ser executadas se uma condição for satisfeita (exemplos: if – else, switch e comandos parecidos);
* **Repetições:** sequencias que devem ser executadas se repetindo até que uma condição seja satisfeita(for, while, do-while, etc).

**Programação Orientada a Objetos (POO)**

O POO é baseado no conceito de “objetos”, que podem conter dados na forma de campos (**atributos**) e códigos, na forma de procedimentos (**métodos**). Uma característica é que um procedimento de objeto pode acessar, e geralmente modificar, os campos de dados do objeto com o qual eles estão associados e cada um é {palavra não identificada no slide, sorry} de receber, processar e enviar dados, podendo ser visto como uma “máquina independente”.

Para que uma linguagem possa ser considerada orientada a objeto é preciso atender a quatro tópicos bastante importante, e esses tópicos é o que são chamados de “Os 4 pilares da Programação Orientada a Objeto”. São eles:

* **Abstração**
* **Encapsulamento**
* **Herança**
* **Polimorfismo**

**Orientação a Objetos em .Net**

A importância da **POO** é simples e direta. Tudo em **.Net** é objeto. Mesmo os tipos de dados mais simples são considerados objetos, já que eles também contêm métodos e propriedades. Implicitamente, todo e qualquer tipo ou objeto em **.Net** possui um ancestral comum.

**O que são Classes**

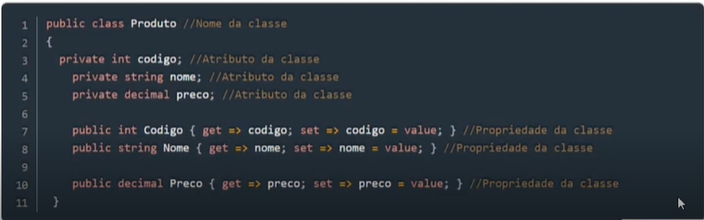
Pode ser considerado como se fosse um molde para o objeto, contendo dentro de si as principais informações para a sua criação. Define atributos e métodos comuns que serão compartilhados por um objeto.

**UML – Linguagem Universal de Modelagem**. A UML define alguns diagramas aonde podemos modelar uma classe.

Exemplo de aplicação UML:

|  |  |
| --- | --- |
| **Produto** | Nome da Classe |
| **-** código: int | Atributos |
| - nome: string |
| - descrição: string |
| - preco\_compra: decimal |
| - preco\_venda: decimal |
| - quantidade\_estoque: int |
| - ativo: bool |
| - data\_cadastro: DateTime |
| + inserir():void | Métodos |
| + alterar():void |
| + excluir():void |
| + pesquisar():void |

Implementação dessa tabela em código abaixo:



**O que são os objetos?**

Considera-se um objeto tudo aquilo que em geral possui atributos, comportamentos e um estado. A classe em si, é um conceito abstrato, como um molde, que se torna concreto e palpável através da criação de um objeto.

**Objetos**

Na programação o objeto é uma **instanciação (criação a partir)** de uma classe. Para criarmos ou instanciarmos objetos de uma determinada classe, utilizamos o operador **new**.

Exemplo:

|  |  |  |
| --- | --- | --- |
| 1 | Produto obj = | new Produto ( ); |

**Visibilidade**

A visibilidade é importante para a proteção do código e suas funcionalidades, define quem pode alterar cada lado dos trechos de código em três principais níveis:

* **Pública**: representada pelo símbolo “+”
* **Privada**: representada pelo símbolo “-”
* **Protegida**: representada pelo símbolo “#”

De forma completa se encontram os modificadores:

|  |  |
| --- | --- |
| Modificador | Significado |
| public | Sem limitação de acesso |
| protected internal | Acesso limitado à própria classe, às classes derivadas e ao próprio assembly. |
| protected | Acesso limitado à própria classe, às classes derivadas. |
| internal | Acesso limitado ao próprio assembly. |
| private | Acesso limitado à própria classe. |

Esse encapsulamento de atributos e métodos impede o chamado vazamento de escopo, onde um atributo ou método é visível por alguém que não deveria vê-lo, como outro objeto ou classe.

Isso evita a confusão de usos de variáveis globais no programa, deixando mais fácil de identificar em qual estado cada variável vai estar a cada momento do programa, já que a restrição de acesso nos permite identificar quem consegue modifica-la.

Valor, Referência e Ponteiros

**Tipos por Valor**

O C# tem duas grandes categorias de tipos: por valor e por referência. Os tipos por valor são gerenciados diretamente e tem as seguintes características principais.

* Todos os tipos de dados numéricos
* Não precisam ser inicializados com o operador **new**
* A variável armazena o valor diretamente.
* A atribuição de uma variável a outra copia o conteúdo, criando efetivamente outra cópia da variável.
* Não podem conter o valor null
* Exemplos de variáveis desse tipo são: integers, doubles, floats e char.

Tipo **Inteiros** ou **Integers**

Os tipos inteiros ou Integers tem sempre o mesmo significado, independentemente da implementação.

|  |  |
| --- | --- |
| Tipo | Implementação |
| byte | Inteiro de 8 bits sem sinal (0 a 255) |
| sbyte | Inteiro de 8 bits com sinal (-127 a 128) |
| ushort | Inteiro de 16 bits sem sinal (0 a 65 535) |
| short | Inteiro de 16 bits com sinal (-32 768 a 32 767) |
| uint | Inteiro de 32 bits sem sinal (0 a 4 294 967 295) |
| int | Inteiro de 32 bits com sinal (-2 147 483 648 a 2 147 483 647) |
| ulong | Inteiro de 64 bits com sinal ( 0 a 18 446 774 073 709 551615) |
| long | Inteiro de 64 bits sem sinal ( -9 223 372 036 854 775 808 a 9 223 372 036 854 775 808) |

**Double** e **Float**

Os números de ponto flutuante são bastante convencionais, as operações de ponto flutuante não geram erro.

A precisão do double é de 15 dígitos decimais. (após a vírgula,óbvio)

A precisão do float é de 7 dígitos decimais

**Caracteres**

Em C#, todos os caracteres (char) são armazenados no padrão Unicode e usam 16 bits de caractere. O Unicode permite armazenar os caracteres de todas as línguas vivas (Como o grego, japonês, chinês e coreano.) e algumas mortas (como sânscrito).

Tipos dor referência

Um tipo por referência armazena uma referência a seus dados. Os tipos de referência incluem o seguinte:

* Duas variáveis poderem conter a referência a um mesmo objeto.
* Operações em uma afetarem a outra.
* Todas as matrizes, mesmo que seus elementos sejam de tipos de valor
* Exemplos de tipos por referência: Strings, classes e arrays.

**Strings**

Semelhante ao char, Strings são variáveis do tipo texto. São uma sequência de caracteres, geralmente utilizadas para representar palavras. Frases e textos de um programa.

String aceita até 1 Giga de caracteres.

As Strings são consideradas imutáveis e não podem ser alteradas depois de criadas. Quando você efetua uma operação qualquer, como por exemplo, concatenar um caractere, você esta na verdade criando uma nova String e descartando a anterior.

**Classes**

Como visto anteriormente é um tipo definido pelo usuário e correspondem a uma class. As classes são sempre derivadas de object e podem conter campos. Métodos e propriedades. Uma classe pode derivar de uma única outra classe, e também de várias interfaces.

**Arrays**

Um array (matriz) é uma lista de valores onde todos os valores no grupo são referenciados pelo nome da matriz e o índice atribuído ao valor específico na matriz.

**Tipo Ponteiro**

Um ponteiro ou apontador é um tipo de dado de cujo valor se refere diretamente a um outro valor alocado em outra área da memória, através de seu endereço.

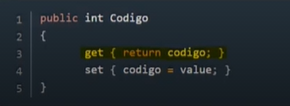
**O que são Métodos?**

Os métodos determinam o comportamento dos objetos de uma classe e são capazes de controlar o estado da instância. São funções que realizam tarefas. Eles podem ou não retornar valores, e podem ou não receber parâmetros.

O envio de mensagens (chamadas de métodos) pode alterar o estado de um objeto. Esses métodos temos como difundidos os Getters, Setters e o Constructs.

**Métodos assessores**

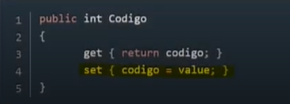
Os **Getters**, ou métodos assessores solicitam o acesso a informações de um determinado produto sem dar acesso diretamente a ele, colocando ali uma barreira de proteção para os dados.



**Métodos Modificadores**

Os **Setters** ou métodos modificadores enviam o pedido de alteração de uma determinada informação de um objeto sem que se altere diretamente o mesmo.

Ainda utilizando do mesmo exemplo o set recebe um valor e o repassa para o atributo.



**Métodos Construtores**

A função do constructo ou método construtor é inicializar ou dar forma a classe. É nele que indicamos os calores dos campos de uma classe. Esses valores podem ser internos (no código) ou externos (passado por parâmetros).

As regras para definir um construtor são:

* O construtor deve ter o mesmo nome da classe
* Não tem tipo de retorno
* É executado apenas um, apenas uma vez, no momento da criação do objeto.
* Não pode ser chamado diretamente
* Deve ser “public”

**O que são Propriedades?**

Sobre

Uma forma mais inteligente de fazer implementações de campos em classes são as propriedades. Propriedades consistem em um par de métodos “get” e “set” que respectivamente servem para recuperar e atribuir o valor a um campo.

Geralmente para casa método existe uma variável dentro da classe que armazena o valor da propriedade.

São mensagens que a classe dispara em determinada situação. Para que o evento funcione, é necessário que um método seja escrito para ser executado quando ocorrer o evento. A classe apenas fica sabendo que existe esse método em tempo de execução.

Para que o mecanismo de eventos funcione, é necessário usar um tipo de estrutura chamado **Delegate**, uma variável que guarda o endereço de uma função. Assim, quando o evento é disparado, essa variável chama a função associada a ela.