**C#**

Declaração = statement

**Comentários**

// your comment

\*/

Commented stuff

/\*

#region region\_name

#endregion

Sempre que algum comando tem () no fim, significa que ele pode receber parâmetros.

**Framework** = conjunto de bibliotecas ou componentes que são usados para criar uma base onde sua aplicação será construída.

**Namespace** = conjunto de funcionalidades

Diferença de Framework, namescape e classe

**Byte** valorMuitoPequeno = 0 até 255

**Short** valorPequeno = -32.768 até 32.768

**Int** valor = 2 bilhões negativos e positivos;

**Long** valorGrande = 9 quintilhões negativos e positivos;

**Float** valor1 = 20.3f; (menor)

**Double** valor2 = 20.3; (médio)

**Decimal** valor3 = 20.3m; (maior)

**String** nome = “texto”;

**Bool** valorVerdade = true;

**Windows Form calcular parcela (dentro do botão executar)**

1. **double** parcela1 = **double**.Parse(text\_parcela1.Text);
2. **double** parcela2 = **double**.Parse(text\_parcela2.Text);
3. **double** resultado = parcela1 / parcela2;
4. label\_resultado.Text = resultado.ToString();

**Prefixo e sufixo “++” “--"**

Se eu colocar dessa forma o ++ não terá efeito, pois o compilador lerá primeiro que deve colocar o resultado como String (e não há como somar valores em strings).

Resultado = 10;

resultado = resultado++.ToString;

Para resolver isso, basta colocar da forma abaixo, pois primeiramente o valor será adicionado em +1 e depois convertido em String.

resultado = (++resultado).ToString;

**ESTRUTURAS DE CONDICIONAMENTO**

**IF ELSE IF**

if (valor == 5)

{  
código;  
}

else if (valor >= 10 && valor == 15 || valor == 20)

{  
código;  
}

else

{  
código;  
}

**TERNÁRIO**

Resultado = valor == 10 ? “SIM” : “NÃO”;

**SWTICH**

switch (valor)

{

case 10:

código;

break;

default:

código;

break;

}

**ESTRUTURAS DE REPETIÇÃO**

**FOR**

For (valor = 0; valor <= 10; valor+=5)

{  
código  
}

**WHILE**

While (valor <= 10)

{  
código  
}

**DO**

Do

{  
código  
} while (valor <= 10)

**FOREACH**

Foreach (char c in frase)

{  
código  
}

**ARRAY**

int[] = valores;

valores = new int[5];

int[] valores = new int[5];

int[] valores = new int[] {10, 20, 30, 40, 50};

**CONTROLE DE STRINGS**

String **frase.ToUpper()** = deixa todos os caracteres em maiúsculo

String **frase.ToLower()** deixa todos os caracteres em minúsculo

String **frase.Length** = pega a quantidade de caracteres na string

String **frase.Substring(**caracter que quer começar a mostrar**,** quantos caracteres quer mostrar depois do índice**)** = pega um pedaço da string

Bool **resultado = frase.Contains(“**letra ou palavra**”)** = retorna um valor booleano dizendo se a letra ou palavra contém na string

Bool **resultado = frase.StartsWith(“**alguma letra ou palavra**”)** = retorna um valor booleano dizendo se a string começa com a letra ou palavra

Bool **resultado = frase.EndsWith(“**alguma letra ou palavra**”)** = retorna um valor booleano dizendo se a string termina com a letra ou palavra

string **frase.Trim()** = remove os espaços entre uma junção strings

string **frase.TrimStart()** = remove os espaços entre uma junção strings apenas no canto de início

string **frase.TrimEnd()** = remove os espaços entre uma junção strings apenas no canto do final

int índice = **frase.IndexOf(**letra ou frase que você está procurando**)** = mostra o valor do index do qual você está procurando

int índice = **frase.LastIndexOf(**letra ou frase que você está procurando**)** = mostra o último valor do index do qual você está procurando

string nome\_completo = **nome.Insert(**index de onde você quer começar a inserir o char ou string**,** nome do char ou string**)** = insere algum char ou string

string resultado = **frase.Replace(**char ou caracteres que quer mudar**,** char ou caracteres novos**)** = muda alguma parte de alguma string

string items = “joão, Paulo, marcos, tulio”;string[]nomes = **items.Split(**aquilo que vai separar cada string**)**Console.WriteLine(nomes[3])= o resultado será “marcos”

**CLASSES**

Primeiro se deve criar uma nova classe.

**Class nomeDaClasse   
{  
}**

Após isso, é possível colocar variáveis dentro da classe:

**Class** **nomeDaClasse   
{**  
 public string nome;  
 public int idade;  
 public string cor\_cabelo;  
**}**

Agora, dentro de outra classe é possível chamar a classe criada:

nomeDaClasse variável = new nomeDaClasse(); 🡨 **CRIAÇÃO DE UM NOVO OBJETO**  
variável.nome = “João”;  
variável.idade = 42;  
variável.cor\_cabelo = “Preto”;  
  
Console.WriteLine(“O nome é: ” + variável.nome);

As classes podem ser úteis para criar objetos onde você não precisa ficar redeclarando uma variável toda vez na sua aplicação principal.

**Class** **nomeDaClasse** 🡨 AQUI ESTAMOS NA NOSSA CLASSE CRIADA **{**  
 public string **nome**;  
 public int **idade**;  
  
 public void **Apresentacao()**  🡨 ISSO É UM MÉTODO QUE CHAMARÁ OUTRO  
 { MÉTODO, PORÉM PRIVADO  
 Console.WriteLine(**ConstruirNomeCompleto());**  
 }  
  
 private string **ConstruirNomeCompleto()** 🡨 ISSO É UM MÉTODO PRIVADO QUE SÓ PODE  
 { SER ACESSADO NA CLASSE nomeDaClasse  
 string identificacao = “Meu nome é ” nome + “ e tenho ” + idade + “anos.”;   
 return identificação;  
 }

AQUI ESTAMOS NA NOSSA APLICAÇÃO PRINCIPAL  
  
nomeDaClasse variável = new nomeDaClasse();  
variável.nome = “João”;  
variável.idade = 42;  
variável.**Apresentacao();**  🡨 ESTAMOS CHAMANDO O MÉTODO DA CLASSE QUE CRIAMOS PARA  
 MOSTRAR O QUE DEFINIMOS

**ACCESS MODIFIERS**

**Public** = é acessível para todas as outras classes.

**Private** = é acessível apenas para os métodos e classes que estão dentro da classe da qual o método ou classe privada está inserida.

**PARÂMETROS**

Quando se cria um método, é possível passar parâmetros para ele dentro dos parênteses:

public int Subtracao(**int parcela1, int parcela2**)  
 {  
 **return** **parcela1** **- parcela2**;  
 }

Nós colocamos os parâmetros dentro dos parênteses quando chamamos o método:

cl\_math calculadora = new cl\_math();  
label\_resultado.Text = calculadora.Divisao(**50, 5**).ToString();

**MÉTODOS COM VALORES PRÉ-DEFINIDOS**

É possível que, dentro de um método, já se estabeleça um valor previamente, sem a necessidade de alguma atribuição anterior:

public int Adicao(**int valor = 10**)  
 {  
 valor += 10; return valor;  
 }

int v = 50;  
cl\_minhaclasse teste = new minhaclasse();  
**v = teste.Adicao()**  
label\_resultado.Text = v.ToString() 🡨 O VALOR MOSTRADO SERÁ 20

Também é possível mesclar os valores passados por parâmetros para dentro dos métodos junto dos valores pré-definidos dentro dos métodos:

public int Adicao(**int x, int valor = 10**) 🡸 TODOS OS VALORES PRÉ-DEFINIDOS DEVEM FICAR  
 { APÓS OS VALORES PASSADOS POR PARÂMETROS  
 valor = valor + x; return valor;  
 }

v = teste.Adicao**(300)** 🡨 SÓ É NECESSÁRIO PASSAR O VALOR NÃO PRÉ-DEFINIDO

**PASSAR ALGUM DADO POR VALOR OU POR REFERÊNCIA**

**Por valor:** Quando se passa um dado por valor para algum método dentro de alguma classe, o que é passado é apenas o valor em si da variável, e não a variável em si. O valor pego pelo método é um parâmetro atribuído que irá ser utilizado dentro daquele método em uma variável local, e posteriormente, ao fim da execução, retornará apenas o novo valor para a variável que estava passando o valor primeiramente.

**Por referência:** Quando se passa um dado por referência para algum método dentro de alguma classe, o que é passado é a própria variável em si. Dessa forma, qualquer alteração feita no valor mudará o valor da variável, mesmo que o método não retorne nenhum valor ao fim.

**CAMPOS E PROPRIEDADES DE UMA CLASSE (SET E GET)**

**Campo:** é uma variável de qualquer tipo declarada diretamente dentro de uma classe ou estrutura. Em geral campos são declarados com visibilidade privada ou protegida. Os campos são escritos com um underline na frente, para melhor identificação.

**Propriedade:** é um membro de uma classe ou estrutura com métodos específicos para acessar o campo que está escondido. Propriedades permitem à uma classe expôr uma maneira de definir seus valores enquanto esconde sua implementação.

**UTILIZAÇÃO DO SET**

1. **class** teste
2. {
4. **private string** \_nome; //ISSO É UM CAMPO
6. **public** **string** Nome //ISSO É UMA PROPRIEDADE
7. {
8. **set**
9. {
10. \_nome = value;
11. }
13. }
15. **public** **void** Apresentacao()
16. {
17. System.Windows.Forms.MessageBox.Show(\_nome);
18. }
20. }
21. **private** **void** button1\_Click(**object** sender, EventArgs e)
22. {
23. teste name = **new** teste();
24. name.Nome = "ManoRusso";
25. name.Apresentacao();
26. }

Neste caso, o que ocorreu foi que a string “ManoRusso” foi passada para a propriedade Nome, e dentro desta propriedade foi utilizado o mecanismo de entrada “set”, que definiu o valor do campo privado “\_nome” com o valor que foi passado para a propriedade Nome através da palavra reservada “value”.

**UTILIZAÇÃO DO GET**

1. **class** teste
2. {
4. **private string** \_nome; //ISSO É UM CAMPO
6. **public** **string** Nome //ISSO É UMA PROPRIEDADE
7. {
8. **get**
9. {
10. \_nome = "Meu nome";
11. **return** \_nome;
12. }
13. }
15. **public** **void** Apresentacao()
16. {
17. System.Windows.Forms.MessageBox.Show(\_nome);
18. }
20. }
21. **private** **void** button1\_Click(**object** sender, EventArgs e)
22. {
23. teste name = **new** teste();
24. **string** x = name.Nome;
25. name.Apresentacao();
26. }

Neste caso, o que ocorreu foi que na variável string “x” ocorreu a atribuição do valor que estava na propriedade Nome, que, neste caso, a propriedade buscou o valor através do mecanismo de saída get, adquirindo o valor do campo privado \_nome, que era a string “Meu nome”. Após buscar o valor, a propriedade Nome retornou o valor de \_nome para a string “x”.

**CONSTRUTORES E OVERLOADING**

**Constructor (construtor):** o (pseudo-)método construtor determina que ações devem ser executadas quando há a criação de um objeto. Em C#, o construtor é definido como um método cujo nome deve ser o mesmo nome da classe e sem indicação do tipo de retorno (nem mesmo void). O construtor é unicamente invocado no momento da criação do objeto através do operador “new“.

Toda classe tem pelo menos um construtor sempre definido. Se nenhum construtor for explicitamente definido pelo programador da classe, um construtor padrão, que não recebe argumentos, é incluído para a classe. No entanto, se o programador da classe criar pelo menos um método construtor, o construtor padrão não será criado automaticamente.

O construtor pode receber argumentos, como qualquer método. Usando o mecanismo de Overloading, mais de um construtor pode ser definido para uma classe.

1. **class** teste
2. {
3. **string** \_nome;
4. **int** \_idade;
6. **public** teste(**string** nome, **int** idade)
7. {
8. \_nome = nome;
9. \_idade = idade;
10. }
12. **public** **void** Apresentar()
13. {
14. System.Windows.Forms.MessageBox.Show(\_nome + " tem " + \_idade + " de idade.");
15. }
16. }
17. **private** **void** button1\_Click(**object** sender, EventArgs e)
18. {
19. teste novo = **new** teste("Matheus", 21);
20. novo.Apresentar();
21. }

O construtor é declarado na linha 6. É possível declarar novos construtores caso eles usem assinaturas distintas das já existentes.

**Overloading (sobrecarga):** este mecanismo possibilita que dois métodos de uma classe tenham o mesmo nome, desde que suas assinaturas sejam diferentes. Tal situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método. Nesse caso, diz-se que ocorre a ligação prematura (early binding) para o método correto.

1. **public** **void** Apresentar()
2. {
3. System.Windows.Forms.MessageBox.Show(\_nome + " tem " + \_idade + " de idade.");
4. }
6. **public** **void** Apresentar(**string** parabens)
7. {
8. System.Windows.Forms.MessageBox.Show(\_nome + " tem " + \_idade + " de idade. Isso é " + parabens);
9. }
10. **private** **void** button1\_Click(**object** sender, EventArgs e)
11. {
12. teste novo = **new** teste("Matheus", 21);
13. novo.Apresentar();
14. novo.Apresentar("legal!");
15. }

**Assinatura de tipo:** jeito de identificar um método de forma única. Ela define as entradas e saídas para uma função, sub-rotina ou método. Uma assinatura de tipo inclui o número de argumentos, os tipos de argumentos e a ordem dos argumentos contidos por uma função.

**ENUMERAÇÕES (ENUM)**

Ele serve para substituirmos constantes nomeadas que são relacionadas mas ficam “perdidas” no código. A palavra-chave enum é usada para declarar uma enumeração, um tipo distinto que consiste em um conjunto de constantes. O Enum é um tipo de valor e não pode herdar ou ser herdado.

Para declarar um enum, escrevemos:

1. **public** **enum** DiasDaSemana
2. {
3. segunda, terça, quarta, quinta, sexta, sábado, domingo
4. }

Agora, para acessa-lo, podemos criar uma nova instância da classe teste (onde está inserido nosso enum DiasDaSemana):

1. **private** **void** button1\_Click(**object** sender, EventArgs e)
2. {
3. teste novo = **new** teste();
4. novo.Apresentar(teste.DiasDaSemana.segunda);
5. }

O método “Apresentar” que está inserido dentro da nossa classe teste chama a enum DiasDaSemana, sendo que o parâmetro “dia” corresponderá ao dia selecionado quando o método é chamado no nosso programa principal (código acima).

1. **public** **void** Apresentar(DiasDaSemana dia)
2. {
3. System.Windows.Forms.MessageBox.Show("O dia escolhido foi: " + dia);
4. }

**CLASSES ESTÁTICAS**

As classes estáticas são aquelas que contêm apenas membros estáticos, além de que elas não podem ser instanciadas, isto é, não podem criar objetos. Todas as propriedades que forem alteradas dentro da classe estática terão efeitos globais para em todos os campos que a utilizem. Se deve evitar a criação de classes estáticas quando há necessidade de instanciação.

Para definir uma classe estática nós escrevemos:

1. **public** **static** **class** ClsMinhaClasseEstatica
2. {
4. }

**Diferença entre classes comuns e classes estáticas:** para uma classe normal ser utilizada ela deve ser instânciada primeiramente (deve ser criado um novo objeto), já as classes estáticas não, pois não há possiblidade da criação de objetos, isto é, ela é acessada diretamente através de modificações de suas propriedades, e qualquer alteração feita em suas propriedades acarreta em uma mudança permanente de suas propriedades, diferente das classes normais que o que é modificado é simplesmente o objeto em si.

Aqui temos uma classe estática criada, junto de uma propriedade (Nome) e um método (Apresentar), ambos – necessariamente – declarados como estáticos

1. **public** **static** **class** ClsMinhaClasseEstatica
2. {
3. **public** **static** **string** Nome; //PROPRIEDADE DA CLASSE ESTÁTICA
5. **public** **static** **void** Apresentar() //MÉTODO DA CLASSE ESTÁTICA
6. {
7. System.Windows.Forms.MessageBox.Show("Seu nome é: " + Nome);
8. }
9. }

Aqui temos nosso programa principal alterando o valor da propriedade estática Nome e chamando o método estático Apresentar (detalhe que começamos escrevendo “teste” pois a nossa classe estática foi escrita dentro do ficheiro teste.class):

1. **private** **void** button1\_Click(**object** sender, EventArgs e)
2. {
3. teste.ClsMinhaClasseEstatica.Nome = "Matheus";
4. teste.ClsMinhaClasseEstatica.Apresentar();
5. }

**CONSTANTES**

É um valor impossível de ser alterado, é o contrário de uma variável. Não é possível declarar constantes sem um valor, uma vez que não é possível altera-los. Vale destacar que para acessar uma constante a partir de um objeto instanciado, é necessário utilizar o get, enquanto isso não é necessário nas classes estáticas. Para se declarar constantes escrevemos:

const int NOMEDACONSTATE = valor;

**CLASSES PARCIAIS**

As classes parciais nada diferem das classes comuns, a denominação “partial” é simplesmente para “partir” a mesma classe em vários pedaços, assim podendo melhorar a organização do código, colocando as propriedades em um arquivo e os métodos em outro.

**HEREDITARIEDADE**

Ela permite que você defina uma classe filha que reutiliza (herda), estende ou modifica o comportamento de uma classe pai. A classe cujos membros são herdados é chamada de classe base. A classe que herda os membros da classe base é chamada de classe derivada. Para uma classe herdar outra, basta colocar ao nome da classe “ : ” e após os dois pontos colocar o nome da classe a ser herdada.

class cl\_carro : cl\_veículo

**CLASSES ABSTRATAS**

As classes abstratas são classes que não podem ser instanciadas, uma vez que servem apenas para serem super classes (classes base), isto é, existem apenas com o propósito de serem herdadas. Para definir uma classe como abstrata, basta colocar antes de “class” a palavra “abstract”.

abstract class cl\_veiculo

**MÉTODOS ABSTRATOS**

Os métodos abstratos são métodos que não possuem qualquer código dentro de uma classe base, e dentro das classes derivadas eles terão que ser necessariamente declarados junto do comando “override”. Mesmo que as classes derivadas usem o mesmo nome de um método que se encontra na classe base, eles serão diferentes com base no código implementado dentro das classes derivadas.

**VIRTUAL**

Colocar o virtual em um método dentro de uma classe base significa que o método definido como virtual poderá ser executado da forma como está na classe base ou então poderá ser executado com o código que está inserido na classe derivada que chamou tal método (através do override).

Código da classe base:

1. **public** **virtual** **void** Parar()
2. {
3. Console.WriteLine("O veículo parou");
4. }

Código da classe derivada carro:

1. **public** **override** **void** Parar()
2. {
3. Console.WriteLine("O carro parou");
4. }

Código da nossa classe principal que chamou ambos os métodos:

1. cl\_carro carro1 = **new** carro();
2. cl\_bicicleta bicicleta1 = **new** bicicleta();
4. carro1.Parar();
5. bicicleta1.Parar();

O resultado que aparecerá no console será:  
“O carro parou  
O veículo parou”

Ou seja, **a diferença de um método virtual para um método abstrato**, é que no primeiro não é necessário sempre o declarar sempre dentro de uma nova classe derivada, é sempre que este método virtual for chamado dentro do código principal da nossa aplicação, será apresentado o código presente na classe base. Enquanto isso, no segundo sempre será necessário declarar o método dentro de uma nova classe derivada, uma vez que a classe abstrata não possui um “código padrão”.

**CLASSE SELADA (SEALED)**

Basicamente, uma classe selada é uma classe que não pode ser herdada. Isto é, uma classe selada pode herdar outras classes (desde que não seladas), mas nunca ser herdada por outra classe. Para criar uma classe selada basta escrever “sealed.”

**CRIAR MAIS DE UM FORMULÁRIO**

Inicialmente, no Form principal, é preciso criar um botão , o qual ao ser clicado abrirá um novo form. Após, é necessário criar um novo Form na solução (mesmo local onde se cria classes). Feito isso, agora é possível criar um novo objeto a partir deste formulário criado, exatamente igual as classes. Portanto, dentro do Form principal deve-se inserir um botão onde o seu click possuirá a seguinte linha de código: “nomeDoObjeto.ShowDialog();”.

**FECHAR FORMULÁRIOS E ENCERRAR A APLICAÇÃO**

Para fechar formulários que não o principal se usa “this.Close()” (detalhe que o “this” fecha toda a classe). Para encerrar a aplicação em sua totalidade se usa “Application.Exit()”.

**MDICONTRAINER E MDICHILD**

Para criar um formulário MDI, primeiramente se deve alterar as propriedades do formulário para “True” na opção “IsMDIContainer”. Após isso, dentro do código do formulário MDI, se deve criar um objeto a partir do formulário que irá aparecer dentro do formulário MDI (este formulário que ficará dentro do formulário MDI é chamado de MDIChild), e então colocar o formulário como child do formulário principal, para isso se usa o comando “nomedoformulario.MdiParent = this;”. Vale ressaltar que para mostrar o formulário se precisa do comando “nomedoformulario.Show();”.

Outra forma de realizar este mesmo processo é através de um parâmetro passado na criação do objeto:

**FORMULÁRIO PRINCIPAL**

1. **private** **void** frmMain\_Load(**object** sender, EventArgs e)
2. {
3. frmChild f = **new** frmChild(**this**);
4. f.Show();
5. }

**FORMULÁRIO CHILD**

1. **public** frmChild(Form frmParent)
2. {
3. InitializeComponent();
4. **this**.MdiParent = frmParent;
5. }

**PS:** .MdiParent significa “é parente de...”.

**MDICHILD APARECER CENTRALIZADO NO MDIPARENT**

Para fazer isso, é necessário pegar o tamanho da janela do formulário pai (preferencialmente maximizado), dividi-lo por dois, e então pegar o tamanho do formulário filho e dividi-lo por 2 também, assim fazendo com que ele fique centralizado:

1. **public** frmChild(Form frmParent)
2. {
3. InitializeComponent();
4. **this**.MdiParent = frmParent;
5. **this**.Location = **new** Point(frmParent.DisplayRectangle.Width / 2 - **this**.DisplayRectangle.Width / 2, frmParent.DisplayRectangle.Height / 2 - **this**.DisplayRectangle.Height / 2);
6. }

**.Location = new Point(x, y); =** Este comando pega dois parâmetros, os quais são a largura e altura.

**.DisplayRetangle.Width/Height =** Este comando pega a largura ou altura de algum formulário.

**UTILIZAÇÃO DE DATAS E HORAS**

É possível criar uma data específica com:

“DateTime objectname = new DateTime();”

Ou então é possível pegar a data atual com:

“DateTime.Now”

**UTILIZAÇÃO DE CORES**

Caixa.BackColor = Color.CorDesejada;

Caixa.BackColor = Color.FromArgb();

**CRIAÇÃO E LEITURA DE FICHEIROS DE TEXTO**

StreamWriter ficheiro = new StreamWriter(@"C:\file.txt", append, Encoding.Default);

O comando .Dispose é utilização para parar um processo (retira-lo da memória)

O comando .Focus faz com que o cursor vá para o determinado local escolhido.

O .EndOfStream indica se determinado ficheiro chegou ao fim de sua leitura.

**DIRETÓRIOS**

File.Exists()

File.Copy/Move(Local do arquivo, local que se criará a cópia)

File.Delete(Local do arquivo);

Directory.Exists()

Directory.CreateDirectory()

Directory.GetFiles(Caminho, “\*.txt”)

Directory.Delete(Local do arquivo, bool)

Path.GetFileName()

O asterisco é usado para dizer algo como “qualquer coisa escrita”.

**CAIXAS DE DIÁLOGO**

MessageBox.Show(“Mensagem”, “Título”, MessaBoxButtons.Botões, MessageBoxIcon.Icone)

Para mudar o que os botões fazem é necessário coloca-los em uma estrutura condicional.

Dialog.AlgumaCoisa

OpenFileDialog objeto = new OpenFileDialog();

Objeto.Title = “Título”

Objeto.InitialDirectory = @”C:\temp”

Objeto.Filter = “Jpeg Images (\*.jpg)|&.jpg|BMP Images (\*.bmp)|\*.bmp”

SaveFileDialog

FolderBrowserDialog

ColorDialog

**COLEÇÕES E LISTAS**

List<tipo> NOME = new List <tipo>();

List<tipo> NOME = new List<tipo>() { 10, 20, 30 };

HashSet<tipo> NOME = new HashSet <tipo>();

Dictionary<int, string> NOME = new Dictionary<int, string>();

Foreach (KeyValuePair<int, string> in item in NOME)

**MANIPULAÇÃO DE FONTES**

Font nomedoobjeto = new Font(“Nome da fonte”, tamanho, FontStyle.Propriedade | FontStyle.Propriedade, GrapichsUnit.Medida);

Label1.Font = nomedoobjeto;

Label1.ForeColor = Color.Red;

**TRY CATCH**

O try é um bloco de código que será executado, e o catch, outro bloco de código que será executado caso haja algum erro no bloco de código no try.

**SQLCE NO C#**

SqlCeConnection objeto1 = new SqlCeConnection();

Objeto1.ConnectionString = @“Data Source=C:zsqlfile.sdf”;

Objeto1.Open();

Objeto1.Close();

SqlCeAdapter objeto2 = new SqlCeAdapter(“SELECT \* FROM Filmes”, ligacao);

DataTable objeto3 = new DataTable();

Objeto2.Fill(objeto3);

**SQLCE**

**Chave primária (Primary Keys, PK):** sob o ponto de vista de um banco de dados relacional, referem-se aos conjuntos de um ou mais campos, cujos valores, considerando a combinação de valores em caso de mais de uma chave primária, nunca se repetem na mesma tabela e, desta forma, podem ser usadas como um índice de referência para criar relacionamentos com as demais tabela do banco de dados (daí vem o nome banco de dados relacional). Portanto, uma chave primária nunca pode ter valor nulo, nem repetição.

**Chave estrangeira (Foreign key):** é o campo que estabelece o relacionamento entre duas tabelas. Assim, uma coluna corresponde à mesma coluna que é a chave primária de outra tabela. Dessa forma, deve-se especificar na tabela que contém a chave estrangeira quais são essas colunas e à qual tabela está relacionada. O banco de dados irá verificar se todos os campos que fazem referências à tabela estão especificados.

**INSTRUÇÃO SELECT**

SELECT \* FROM Pessoas

SELECT nome, idade, sexo FROM Pessoas

SELECT [nome da pessoa], [idade da pessoa] FROM Pessoas

SELECT \*, \* FROM Clientes, Compras

**INSTRUÇÃO INSERT UPDATE DELETE**

INSERT INTO Pessoas VALUES (‘Matheus’, 21, ’masculino’)

UPDATE Pessoas SET idade = ‘22’ *<- todos os valores da tabela seriam alterados*

UPDATE Pessoas SET idade = ‘22’ WHERE nome = ‘Matheus’

UPDATE Pessoas SET nome = ‘Ana’, idade = ‘22’ WHERE nome = ‘Matheus’

DELETE FROM Pessoas WHERE nome = ‘Matheus’

**CLAUSULA WHERE E OPERADORES DE COMPARAÇÃO**

SELECT nome, idade FROM Pessoas WHERE sexo **<>** ‘masculino’ *<- sinal de diferente*

SELECT \* FROM Pessoas WHERE idade **>** 18 *<- sinal de maior*

SELECT \* FROM Pessoas WHERE idade **<** 18 *<- sinal de menor*

SELECT \* FROM Pessoas WHERE idade **>=** 18 *<- sinal de maior ou igual*

SELECT \* FROM Pessoas WHERE idade **<=** 18 *<- sinal de menor ou igual*

SELECT \* FROM Compras WHERE quantidade **\*** preco\_unidade >= 10 *<- utilização de sinais matemáticos*

**CLAUSULA WHERE E AND OU OR**

SELECT \* FROM Pessoas WHERE nome = ‘Matheus’ **AND** idade = 21

SELECT produto FROM Compras WHERE quantidade >= 5 **OR** preco\_unidade < 10

SELECT \* FROM Compras WHERE (produto = ‘Banana’ AND quantidade >= 5) OR (produto = ‘Laranja’ AND quantidade >= 10)

SELECT \* FROM Compras WHERE produto = ‘Banana’ AND (quantidade = 5 OR quantidade = 10)

**CLAUSULA BETWEEN IN E LIKE**

SELECT \* FROM Compras WHERE quantidade BETWEEN 3 AND 7

SELECT \* FROM Compras WHERE id\_cliente IN (1, 5, 7)

SELECT \* FROM Clientes WHERE nome IN (M#) *<- a wildcard % significa “quaisquer caracteres depois”*

SELECT \* FROM Clientes WHERE produto IN (Laranj\_) *<- a wildcard \_ significa “qualquer letra”*

**CLAUSULA ORDER BY E DISTINCT**

SELECT \* FROM Clientes **ORDER BY** nome **ASC** *<- o termo ASC significa “ascending”, isto é, de forma crescente (de A – Z)*

SELECT \* FROM Clientes WHERE id\_clinte >= 5 **ORDER BY** nome **DESC**

SELECT **DISTINCT** produto from Compras *<- será mostrando apenas os produtos diferentes presentes na tabela, nada de repetições*

**CLAUSULA TOP MAX MIN E ALIAS**

SELECT **TOP (5)** FROM Clientes

SELECT **TOP (5)** FROM Clientes ORDER BY id\_cliente DESC

SELECT **MAX**(quantidade) FROM Compras

SELECT **MIN**(quantidade) **AS** quantidademaxima FROM Compras *<- “as” é um alias*

SELECT **MAX**(quantidade) **AS** maiorquantidade FROM Compras WHERE produto = ‘Laranja’

SELECT nome **AS** nomedapessoa, telefone **AS** numerodecelular FROM Clientes

SELECT p**.**\*, c**.**\* FROM Pessoas **p**, Compras **c** WHERE **p.**nome = ‘Matheus’ AND **c.**produto = ‘Laranja’ *<- é possível escrever tudo isso, da mesma forma, sem o ALIAS*

**CLAUSULA INNER JOIN LEFT JOIN E RIGHT JOIN**

SELECT \* FROM Clientes INNER JOIN Compras ON Clientes.id\_cliente = Compras.id\_cliente

SELECT \* FROM Clientes LEFT JOIN Compras ON Clientes.id\_cliente = Compras.id\_cliente WHERE Clientes.nome = ‘Ana’

**SQL FUNÇÕES**

SELECT **AVG**(quantidade) FROM Compras *<- mostra a média*

SELECT **AVG**(quantidade) AS media FROM Compras WHERE produto = ‘Banana’

SELECT **COUNT**(id\_cliente) FROM Clientes *<- mostra a quantidade de registros*

SELECT **SUM**(quantidade) FROM Compras WHERE produto – ‘Maça’ *<- mostra a somatória dos valores*

SELECT **SUM**(quantidade \* preco) FROM Compras *<- ele calcula linha por linha*

**CLAUSULA CREATE TABLE**

CREATE TABLE Encomendas

(

id\_encomenda int not null, primary key,

id\_cliente int not null,

produto nvarchar(50),

quantidade int,

data\_encomenda datetime,

enviado bit,

FOREIGN KEY (id\_cliente) REFERENCES Clientes(id\_cliente) ON DELETE CASCADE

)