**Programação em c# iniciante. Primeiros conceitos.**

**Namespaces:**

- Usado para organizar tipos (classes), permitindo evitar colisões de nomes.

Basicamente um namespace é uma coleção nomeada de classes.

Boa Pratica: todas as classes devem ser definidas dentro de um namespace

**Ex:**

Classe: Console

Nome completo: System.Console

Classe Console (definida pelo usuário)

Namespace MinhasClasses

MinhasClasses.Console

**Diretiva using**

Diretivas using trazem os namespaces para o escopo do seu projeto; desta forma não é necessário qualificar explicitamente os objetos com o respectivo namespace.

Ou seja, a diretiva using inclui outros namspaces no programa.

**Assembly / Montagem**

**-** As classes são compiladas em Assemblies – são arquivos, usualmente com a extensão .dll

As classes mais comuns como System.Console pertecem ao assembly *mscorlib.dll.*

Uma montagem pode conter classes definidas em muitos namespaces, e um namespace pode ocupar vários assemblies.

Para usarmos as classes de um assembly, devemos adicionar uma referencia no projeto, e incluir a diretiva “using” adequada.

**Referencias:**

Uma referencia permite que utilizemos tipos ou classes disponíveis em outros assemblies, tanto do FCL quanto de terceiros

FCL = framework class library

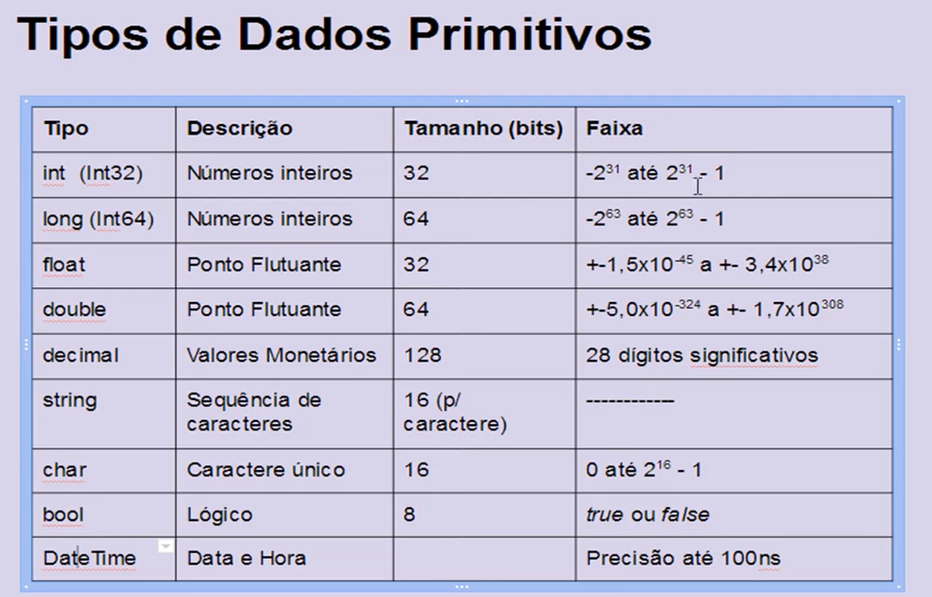
**Tipos de variáveis**

**Declaração: tipo\_variável nome\_variavel**

**Ex:** int num1 = 0;

Str nome = “lucas”

Bool trava = True

****

**Arrays**

Um array (“vetor”) é uma sequencia não-ordenada de itens.

Todos os itens dentro de um array são do mesmo tipo.

Os itens do array são armazenados em um array um bloco contiguo da memoria RAM e são acessados por meio de um número de índice.

**Sintaxe:** tipo[] nome\_array;

Ex: int[] números\_loteria;

Obs.: O tamanho do array não faz parte da declaração.

Podemos criar array de qualquer tipo, tanto de valor quanto de referência.

**Criar uma instancia do array:**

Os arrays são tipos de referencia, independentemente do tipo de seus elementos.

Ou seja, uma variável array faz referencia a um bloco de memoria que armazenara os elementos do array no heap

Assim como com as variáveis de classe, a memoria não é alocada para o array até que criemos uma instancia dele usando o operador new.

O array somente recebe espaço na memória quando a instancia é criada – e é quando especificamos o tamanho do array.

**Preenchendo um array**

Podemos inicializar os elementos de um array fornecendo uma lista de elementos separados por virgulas, entre chaves:

Int[] números\_loteria = new int[6] {2, 23, 34, 12, 29, 44}

Podemos também usar essa sintaxe:

Int[] números\_loteria = {2, 23, 34, 12, 29, 44};

Omitindo a palavra new – o compilador calcula o tamanho do array a partir da lista de elementos inicializadores e gera o código que cria o array.

**Métodos**

Um método é uma sequência de declarações (comandos) que possui um nome de identificação. É similar a uma função ou procedimento.

Um método possui um nome e um corpo onde ficam os comandos que serão executados quando o método for chamado.

Os métodos também podem receber dados para processamento (parametros) e retornar informações.

**Declaração de métodos**

**Sintaxe:**

acesso tipo\_retorno nome\_metodo (parâmetros)

{

corpo do método (variáveis e declarações);

}

Caso estejamos escrevendo um método que não ira retornar nenhum valor, usaremos a palavras *void* no lugar de tipo\_retorno.

Um método deve sempre executar uma operação única, devemos criar um método para cada tarefa.

**Chamado de métodos.**

**Sintaxe:**

resultado = nome\_metodo(argumentos);

Se o método for void (não retorna valores), então a clausula resultado = não será usada:

nome\_metodo(argumentos);

**Parâmetros**

Em C# temos quatro tipos de parâmetros de métodos:

- valor

- ref

- out

- params

**Valor –** Tipo padrão. Os parâmetros de valor passam uma cópia local de si mesmos ao método.

Ou seja, o método usa o parâmetro, porem a cópia original do chamador não é modificada.

O argumento passado deve ser do mesmo tipo do parâmetro, ou ao menos conversível implicitamente.

**Ref –** Prefixando um parâmetro com a palavra ref, o compilador do C# gera um código que passa uma referencia a argumento real em vez de uma cópia do argumento.

Tanto o argumento quanto o parâmetro referenciam os mesmos dados.

*Nesse caso a variável do escopo Main é alterada permanentemente.*

**Overloading de métodos.**

Métodos sobrecarregados são métodos que possuem o mesmo nome e são declarados no mesmo escopo.

A sobrecarga é útil quando precisamos efetuar a operação em tipos diferentes de dados ou conjuntos de informações que variam.

Métodos sobrecarregados podem possuir numero de parâmetros diferente ou ainda tipos de parâmetros diferentes.

O tipo de dados de retorno, contudo, é sempre o mesmo para todos os métodos sobrecarregados.

Um exemplo clássico é o método **WriteLine** da classe **Console**.

**Parâmetros opcionais em métodos.**

- Um parâmetro é considerado opcional quando ele tem um **valor padrão** atribuído a ele.

Todos os parâmetros mandatórios devem ser especificados antes dos opcionais.

O C# usa a posição de cada argumento passado ao método para determinar a quais parâmetros eles se referem(em ordem).

**Argumentos Nomeados.**

É possível passar os argumentos para um método em qualquer ordem.

**Sintaxe:**

metodo(nome\_parametro: argumento)

**Modificadores de Acesso.**

**-** Um modificador de acesso permite determinar o nível de acessibilidade dos membros e tipos em um método, controlando assim como eles podem – ou não – serem acessados por outros métodos ou a partir de outros assemblies.

Os modificadores de acesso em C# são:

- public

- private

- protected

- internal

- protected internal

**Public**

O tipo ou membro **public** pode ser acessado sem restrições por qualquer outro código no mesmo assembly ou em outros assemblies que façam referência a ele.

Portanto, são visíveis pelos métodos de qualquer classe.

**Private**

O tipo ou membro **private** só pode ser acessado por códigos que estejam na mesma classe ou struct.

**Protected**

O tipo ou membro **protected** só pode ser acessado por códigos da classe ou struct ao qual pertencem, ou ainda em uma classe que seja derivada esta classe que contém o modificador.

**Protected internal**

O tipo ou membro **protected internal** pode ser acessado por qualquer código no mesmo assembly, ou de uma classe derivada em outro assembly.

**Modificador static**

Podemos usar a palavra chave static para criamos um método estático, de modo que não seja necessário instanciar a classe para usá-lo.

O modificador static indica que o membro em questão pertence à classe em si, e não às instâncias da classe.

Apenas uma cópia do membro estático existe ne aplicação, mesmo que várias instancias da classe sejam criadas.

**Classes**

São “pacotes” de código que definem tipos de dados, os quais possuem propriedades e operações (**métodos**) em linguagens orientadas a objetos (**OOP**).

As propriedades armazenam valores e características dos dados.

Já os métodos são funções internas às classes, que permitem agir sobre elas para executar ações.

**Objetos**

São instâncias de um tipo (classe) que possuem características e comportamentos.

Podemos criar múltiplas instancias de uma classe.

As propriedades, métodos e eventos de uma classe são alocadas em memória através da instancia de um objeto..

**Vantagens da OOP**

- Reutilização dos códigos

- Modularidade

- Uso mais simples (realístico)

- Código mais limpo e claro.

**Criar classes**

Usamos a palavra chave **class** para definir uma nova classe.

class Caixa

{

double lado;

double Volume()

{

Return lado \* lado \* lado

}

}

**Instanciando uma classe.**

Para acessarmos nossos métodos e dados criados na nossa classe primeira precisamos instanciá-la.

Instanciando a classe Caixa criada anteriormente.

class Program

{

static void Main()

{

Caixa cx; criando o objeto – NomeClasse nomeObj

cx = new Caixa(); chamando construtor – nomeObj = new Nomeclasse()

}

}

**Método Construtor**

Para que possamos usar essa classe ainda é necessário que o campo lado seja inicializado – é inacessível do exterior pois é privado.

Faremos isso usando um Construtor.

Quando usamos a palavra chave **new** para criar um objeto, o runtime constrói o objeto usando a definição da classe.

O runtime toma um pedaço da memória RAM, a preenche com os campos definidos pela classe e então invoca um construtor que realizará as inicializações requeridas.

Um construtor é um método especial que roda automaticamente quando uma classe é instanciada.

Possui o mesmo nome da classe, e pode receber parâmetros, ***mas não retorna valores*** (nem mesmo void!)

Toda classe deve ter um construtor; se um não for escrito, o compilador gerará um automaticamente – um que não faz absolutamente nada.