**Tipos (Guia de Programação em C#)**

**https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/types/#types-of-literal-values**

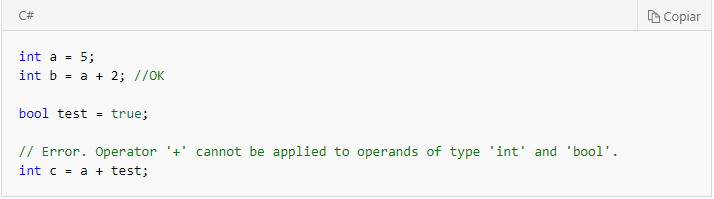
**Tipos, variáveis e valores**

C# é uma linguagem fortemente tipada, por esse motivo todas as variáveis e constantes tem um tipo, assim como cada expressão que é avaliada como um valor, ou seja, cada expressão feita na linguagem C# resulta em um valor existente dentro da linguagem C#. Cada assinatura de método especifica um tipo para cada parâmetro de entrada e para o valor de retorno. A biblioteca de classes .NET define um conjunto define um conjunto de tipos numéricos internos, e também tipos mais complexos que representam uma ampla variedade de constructos lógicos, como sistema de arquivos, as conexões de rede, as coleções e as matrizes de objetos e as datas. Um programa C# típico usa tipos da biblioteca de classes, bem como tipos definidos do usuário que modelam os conceitos que são específicos para o domínio do problema do programa.

As informações armazenadas em um tipo, pode incluir o seguinte:

* O espaço de armazenamento que uma variável do tipo requer.
* Os valores mínimo e máximo que ele pode representar.
* Os membros (métodos, campos, eventos e etc.) que ele contém.
* O tipo base do qual ele herda.
* O local no qual a memória para as variáveis será alocada em tempo de execução.
* Os tipos de operações que são permitidos

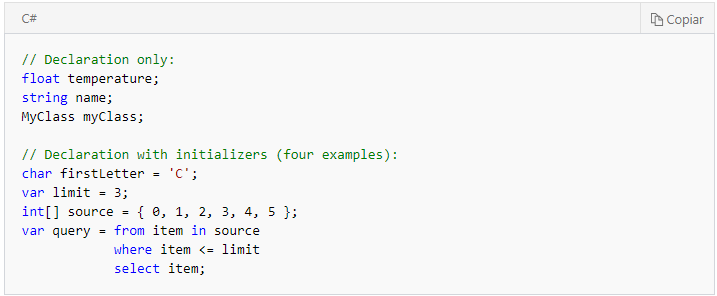
São as informações do tipo que são utilizadas pelo compilador para realizar as validações necessárias, e garantir que todas as operações realizadas no código sejam fortemente tipadas.



O compilador insere as informações de tipo no arquivo executável como metadados. O CLR (Common Language Runtime) usa metadados em tempo de execução para garantir mais segurança de tipos quando aloca e recupera memória.

**Especificando tipos em declarações de variável**

Quando estamos especificando uma variável ou constante num programa, devemos especificar os eu tipo explicitamente ou utilizar a palavra-chave var para permitir que o compilador infira um tipo.





Depois que uma variável é declarada ela não pode ser declarada novamente com um novo tipo e não pode ter um valor atribuído que não seja compatível com seu tipo declarado. No entanto, os valores podem ser convertidos em outros tipos. Uma conversão de tipo que não causa a perda de dados é executada automaticamente pelo compilador. Uma conversão que causa perda de dados requer um cast no código-fonte.

**Tipos internos**

O C# fornece um conjunto padrão de tipos numéricos internos para representar números inteiros, valores de ponto flutuante, expressões boolianas, caracter de texto, valores decimais e outros tipos de dados. Tambem são levados em consideração o tipo string e object internos. Que estão disponíveis para uso em qualquer programa C#.

**Tipos personalizados**

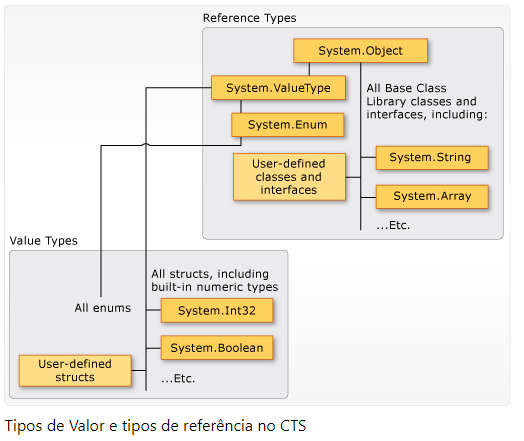
Utilizamos os construtos struct, classe, interface e enum para criar nossos próprios tipos personalizados. A biblioteca de classes do .NET em si é uma coleção de tipos personalizados fornecida pela Microsoft, que podemos utilizar em nossos aplicativos.

**O CTS (Common Type System)**

É importante entender os dois pontos fundamentais sobre o sistema de tipos do .NET

* Ele dá suporte ao conceito de herança. Isso significa que os tipos podem derivar de outros tipos, que são chamados tipos base. O tipo derivado herda (obedencendo as restriç~esod definidas) os me´todos as propriedades e outros membros do tipo base. O tipo base por sua vez também pode ser derivado de algum outro tipo, nessa sigtuaçção, o tipo derivado herdas os membros de ambos os tipos base na sua hierarquia de herança.
* Cada tipo no CTS é definido como um tipo de valor ou um tipo de referência. Isso inclui todos os tipos que já foram criados na biblioteca de classes .NET, ou seja, são tipos que fazem parte do framework, bem como tipos personalizados definidos pelo usuário, aqueles tipos inerentes a cada domínio. Os tipos definidos utilizando a palavra chave struct são tipos de valor, por exemplo, os valores numéricos internos são definidos com struct. Os tipos que são definidos utilizando a palavra chave class são tipos de referência. Ocorre que os tipos de valor e os tipos de referência têm diferentes regras de tempo de compilação e comportamento de tempo de execução diferente.

A imagem abaixo mostra como ser relacionam os tipos de referência e os tipos de valor no CTS.



**Tipos de Valor**

As variáveis do tipo de valor derivam de System.ValueType, e essa por sua vez deriva de System.Object. Os tipos que derivam de System.ValueType apresentam um comportamento especial no CLR (porque o tratamento é espcial?). As variáveis do tipo valor contêm diretamente seus valores, ou seja, eles não tem outra estrutura de armazenamento diferente da memória, a estrutura utilizada, o que significa que a memória é alocada embutida em qualquer contexto em que a variável é declarada. Não há nenhuma alocação de heap separada ou sobrecarga de coleta de lixo para variáveis dos tipos valor.

Há duas categorias do tipo valor

* Struct
  + Tipos numéricos são structs tem propriedades e métodos que você pode acessar
    - byte b = Byte.MaxValue;
  + Mas é possível declarar e atribuir valores a eles como ser fosses tiopios de não agregação
    - byte num = 0xA;   
      int i = 5;   
      char c = 'Z';
* Enum

Os tipos de valor utilizam o modificador sealed, o que impede de algum tipo ser derivado deles,por exemplo, não podmos derivar do tipo System.ValueType. Contudo o struct pode implementar uma ou mais interfaces Podemos conveter um tipo struct emn um tipo de interface.Isso faz com que uma operação de donversçao boxing encapsule o struct dentro de um objeto de tipo de referencia no heap gerenciado. O momento no qual uma opéraç]ã de boxingo ocorre é quando há convertão de um tipo de valor em tipo baseado no System.object com parâmetro de entrada.

Podemos utilizar a palavra-chave struct para criar nossos próprios tipos de valor personalizados. Comumente utilizando o struct como um container para um conjunto de variáveis relacionados.

public struct CoOrds

{

public int x, y;

public CoOrds(int p1, int p2)

{

x = p1;

y = p2;

}

}

A outra categoria de tipos de valor é o enum. Uma enum define um conjunto de constantes integrais nomeadas, conforme o exemplo abaixo com base na Sysmte.IO.FileMode.

public enum FileMode

{

CreateNew = 1,

Create = 2,

Open = 3,

OpenOrCreate = 4,

Truncate = 5,

Append = 6,

}

A vantagem de se utilizar enum é que um nome á muito mais signicativo e intuitivo para leitura do código fonte por humanos do que um número, por esse motivo é melhor utilizar enumerações ao invés de números literais constantes.

Todas as enumerações, herdam de System.Enum, que herdam de System.ValueType. As regras aplicadas a structs também se aplicam a enums.

**Tipos de Referência**

Um tipo que é definido como uma classe, delegate, matriz ou interface é um tipo de referência. Em tempo de exeucção, quando declaramos uma variável do tipo referência, à variável inicialmente está com o valor null até que seja criada explictametno um objeto usando o operador new ou atribuir a ele um objeto que foi criado em outro lugar com o operador new.

Exemplo:

MyClass mc = new MyClass();

MyClass mc2 = mc;

No caso de interfaces, a variável deve ser inicializada com um objeto da classe que implementa a interface.

IMyInterface iface = new MyClass();

Quando o objeto é criado, ele é colocado no heap gerenciado e a variável contém apenas uma referência para o local do objeto. Os tipos no heap gerenciado requerem sobrecarga quando são alocados e quando sção recuperados pelao funcionalidade de gerenciamento automaático de memória, que conhecida como coleta de lixo. Contudo a coleta de lixo é altamente otimizada e na maioria dos cenários não criar problema de desempenho.

Todas as matrizes são do tipo referência, mesmo que os elementos contidos nela forem do tipo valor. As matrizes derivam implicitametno da classe System.Array, mas declaramos e usamos as matrizes com a sintaxe siplifidade fornecida pelo C#.

Exemplo:

// Declare and initialize an array of integers.

int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.

int len = nums.Length;

Os tipos de referência tem total suporta para herança. Ao criar uma classe, podemos herdar de uma classe ou implementar uma interface que são está definida como lacrada (sealed), e outros classes pode herdar das classes criadas e substituir os métodos virtuais.

**Tipos de valores literais**

No C#, valore literais recebem um tipo do compilador, ou seja, mesmo não declarando com o tipo explicitamente, o compilador tem condições de inferir um tipo aquele literal. Podemos especificar como um literal numérico deve ser digitado anexando uma letra ao final do numero. Por exemplo, para especificar que o valor 4.56 deve ser tratado como um float, acrescentamos um “f” ou “F” após o numero: 4.56f. Se nenhuma letra for anexada, o compilador inferirá uma tipo para o literal.

**Tipos Genéricos**

Para alguns casos não precisamos especificar o tipo inicialmente, nesse caso utilizamos parâmetros de tipo que servem como um espaço reservado para o tipo real (o tipo concreto) que o código cliente fornecerá ao criar uma instância do tipo. Esses tipos são chamados de tipos genéricos. Como exemplo, o tipo .NET System.Collections.Generic.List<T> tem um parâmetro de tipo, que por convenção, recebe o nome T. Ao criar uma instância do tipo, você pode espeficicar o tipo dos objetos que a lista conterá, como exemplo, cadeia de caracters, conforme abaixo.

List<string> stringList = new List<string>();

stringList.Add("String example");

// compile time error adding a type other than a string:

stringList.Add(4);

O uso do parâmetro de tipo possibilita a reutilização da mesma classe para conter qualquer tipo de elemento sem precisar converter cada elemento em objeto. As classes de coleções genéricas são chamadas de coleções fortemente tipadas por que o compilador sabe os tipos específico dos elementos da coleção e pode gerar um erro em tempo de compilação se, por exemplo, você tentar adicionar um inteiro ao objeto stringList no exemplo anterior.

**Tipos implícitos, tipos anônimos e tipos que permitem valor nulo**

Conforme mencionado antes, podemos declarar implicitamente uma variável local (mas não os membros de classe) usando a palavra-chave var. A variável recebe um tipo em tempo de compilação, mas o tipo é fornecido pelo compilador.

Em alguns casos, é inconveniente criar um tipo nomeado para conjunto simples de valores relacionados que você não pretende armazenar ou transmitir fora dos limites de método. Você pode criar tipos anônimos para essa finalidade.

Os tipos comuns de valor não podem ter um valor nulo. No entanto, você pode criar tipos de valor anulável afixando uma ? após o tipo. Por exemplo, int? é um tipo int que também pode ter valor nulo. No CTS, os tipos anuláveis são instâncias do tipo struct genérico System.Nullable<T>. Os tipos que permitem valor nulo são especialmente úteis quando você está passando dados entre bancos de dados nos quais valores numérciso podem ser nulos.

**Tipos de Valor**

<https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/value-types>

Os tipos de valor são divididos em duas categorias principais: Enum’s e struct’s.

Os structs também são divididos em categorias, conforme a lista abaixo.

* Tipos numéricos
  + Tipos integrais
  + Tipos de ponto flutuante
  + Decimal
* Bool
* Structs definidos pelo usuário

**Principais recursos do tipo de valor**

As variáveis que são baseados diretamente em tipos de valor contem valores. Atribuir uma variável de tipo de valor outra, copia o valor contido. Isso é diferente na atribuição de variáveis nos tipos de referência, que copiam uma referência para o objeto, mas não o próprio objeto.

Os tipos de valor são derivados da System.ValueType.

Diferente dos tipo de referência, não podemos derivar um novo tipo com base em um tipo valor. No entanto, igual aos tipos de referência, é possível implementar interfaces com structs.

Diferente dos tipos de referência, um tipo valor não pode conter o valor null, contudo há um recurso de tipo que permitem valor nulo permite que os tipos de valor seja atríbuido com o valor null.

Cada tipo de valor tem um construtor padão implícito que inicializa o valor padrão desse tipo.

**Principais recursos dos tipos simples**

Todos os tipos simples – integrantes da linguagem C# - são aliases dos tipos de sistemas .NET Framework. Por exemplo, o tipo int é um alias de System.Int32

As expressões constantes, cujos os operandos são todos contantes de tipo simples, são avalidados em tempo de compilação.

Os tipo simples podem ser iniciallizda com o uso de literais.

struct (Referência de C#)

<https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/struct>

Normalmente utilizamos struct para encapsular pequenos grupos de variáveis relacionadas, tais como coordenadas de um retângulo ou características de um item de inventário.

Os structs também podem conter construtores, constantes, campos, métodos, propriedades, indexadores, operadores, eventos e tipos aninhados, embora, se muitos dessses membros forem necessários, devemos considera o tipo definido como struct em um class.

Os structs podem implantar uma interface, mas não podem herdá-la de outra struct. Por esse motivo os membros de struct não podem ser declarados como Protected.

# enum (Referência de C#)

<https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/enum>