**Resumo da Semana**

**Namespaces:**Co aumento do sistema, é preciso muito organização entre os arquivos e saber de qual escopo faz parte cada classe, o grande problema que surge com os sistemas grandes é a organização de todas as suas classes. Para evitar que o sistema fique caótico, podemos agrupar as classes por características comuns e dar um nome para cada um desses grupos. Isto é, agruparíamos um conjunto de classes em um espaço em comum e lhe daríamos um nome, como por exemplo StoneCo.CSharpEvolution.WeeklyChallenge. Esse espaço definido por um nome é chamado de namespace.

Segundo a convenção de nomes adotada pela Microsoft (<https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-1.1/893ke618(v=vs.71)>), os namespaces devem ter a forma: NomeDaEmpresa.NomeDoProjeto.ModuloDoSistema.

**Classes**Uma classe é uma forma de definir um tipo de dado, ela é formada por dados e comportamentos.

Para definir os dados, são utilizados os atributos, e para definir o comportamento são utilizados métodos. Depois que uma classe é definida podem ser criados diferentes objetos que utilizam a classe.

O código da classe Conta, por convenção, deve ficar dentro de um arquivo com o mesmo nome da classe.

**Structs**

Os structs são particularmente úteis para estruturas de dados pequenas que têm semântica de valor. Números complexos, pontos em um sistema de coordenadas ou pares chave-valor em um dicionário são exemplos de structs. O uso de structs, em vez de classes para estruturas de dados pequenas, pode fazer uma grande diferença no número de alocações de memória que um aplicativo executa.Ao contrário das classes, as estruturas são tipos de valor e não precisam de alocação de heap.

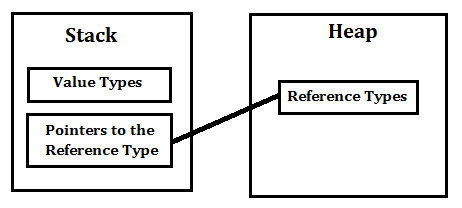
Os construtores struct são invocados com o operador new, semelhante a um construtor de classe. Porém, em vez de alocar dinamicamente um objeto no heap gerenciado e retornar uma referência a ele, um construtor de struct simplesmente retorna o valor do struct (normalmente em um local temporário na pilha), e esse valor é, então, copiado conforme a necessidade.

**Principais Tipos Primitivos**

**Principais Tipos de Referência**

Há dois tipos em C#: tipos de referência e valor. Variáveis de tipos de referência armazenam referências em seus dados (objetos) enquanto que variáveis de tipos de valor contém diretamente seus dados. Com tipos de referência, duas variáveis podem fazer referência ao mesmo objeto; portanto, operações em uma variável podem afetar o objeto referenciado pela outra variável. Com tipos de valor, cada variável tem sua própria cópia dos dados e as operações em uma variável não podem afetar a outra (exceto no caso das variáveis de parâmetros in, ref e out. Confira o modificador de parâmetro in, ref e out).

Quando você atribui um novo valor a uma variável de um tipo de valor, esse valor é copiado. Quando você atribui um novo valor a uma variável de um tipo de referência, a referência é copiada, não o objeto.



**Tipo de Acessibilidade (Classes e métodos)**.

Use os modificadores de acesso, public, protected, internal ou private, para especificar um dos níveis de acessibilidade declarada a seguir para membros.

Private  
O tipo ou membro pode ser acessado somente pelo código na mesma classe ou struct.

Protected  
O tipo ou membro pode ser acessado somente pelo código na mesma classe ou em uma classe derivada dessa classe.

Internal  
O tipo ou membro pode ser acessado por qualquer código no mesmo assembly, mas não de outro assembly.

protected internal   
O tipo ou membro pode ser acessado por qualquer código no assembly no qual ele é declarado ou de uma classe derivada em outro assembly.

private protected   
O tipo ou membro pode ser acessado somente dentro de seu assembly de declaração, por código na mesma classe ou em um tipo que é derivado dessa classe.

Os modificadores de acesso não são permitidos em namespaces. Namespaces não têm nenhuma restrição de acesso.

Dependendo do contexto no qual ocorre uma declaração de membro, apenas algumas acessibilidades declaradas são permitidas. Se não for especificado nenhum modificador de acesso em uma declaração de membro, uma acessibilidade padrão será usada(para classes será o private.

**Construtores**

Os construtores são tipos especiais de métodos usados ​​para criar e inicializar objetos. É através deste tipo especial de método, chamado de construtor, que você cria instâncias de uma classe.

Assim, os construtores permitem que criem objetos da classe e defina valores padrão, limite a instanciação e grave códigos flexíveis e fáceis de ler.

Pontos importantes para lembrar sobre construtores:

* Construtor de uma classe deve ter o mesmo nome que o nome da classe na qual ele reside.
* Um construtor não pode ser abstrato, final, estático e sincronizado.
* Dentro de uma classe, você pode criar apenas um construtor estático.
* Um construtor não tem nenhum tipo de retorno, nem mesmo anulado.
* Um construtor estático não pode ser um construtor parametrizado.
* Uma classe pode ter qualquer número de construtores.
* Os modificadores de acesso podem ser usados na declaração do construtor para controlar seu acesso, ou seja, qual outra classe pode chamar o construtor.

Tabela com os valores definidos pelo construtor padrão para os tipos por valor:

|  |  |
| --- | --- |
| **Tipo de valor** | **Valor padrão** |
| [**bool**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/bool) | FALSE |
| [**byte**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/byte) | 0 |
| [**char**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/char) | '\0' |
| [**decimal**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/decimal) | 0M |
| [**double**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/double) | 0,0D |
| [**enum**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/enum) | O valor produzido pela expressão (E)0, em que E é o identificador de enumeração. |
| [**float**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/float) | 0,0F |
| [**int**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/int) | 0 |
| [**long**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/long) | 0L |
| [**sbyte**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/sbyte) | 0 |
| [**short**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/short) | 0 |
| [**uint**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/uint) | 0 |
| [**ulong**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/ulong) | 0 |
| [**ushort**](https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/ushort) | 0 |

**Destrutores**Métodos Destrutores servem para liberar a memória alocada dinamicamente pela classe, para eliminar as referências a ela, quando não existir.

Em Linguagens de programação que tem o Garbage Collector, não se faz necessário o uso de métodos destrutores, pois o Garbage Collector se encarrega de fazer isso.

A necessidade de cria-los, é em casos em que a linguagem não tem o Garbage Collector, e se faz necessário destruir a classe após sua utilização, para que não fique ocupando mémoria.

Em linguagens que tem o Garbage Collector, só se faz necessário o uso, quando da utilização de recursos não gerenciados.

Nesses casos o Garbage Collector não sabe como liberar e limpar o recurso não gerenciado.

Para designar um destrutor é colocado um “**~**” na frente do nome da classe a ser destruida. Exemplo: ~ComplexCalculator().

**Condicionais**   
**Loops**   
**Interrupção de Loops**   
**Métodos**

**Descreva as principais funções e o que encontramos do ponto de vista macro (Só o que contém, de forma mais abrangente) nos namespaces e destaque no mínimo 03 classes que podemos trabalhar em cada um deles:**System.Text,   
System.Threading,   
System.Linq,   
System.IO,   
System.Data,   
System.Security,   
System.Diagnostics,   
System.Runtime,   
System.Globalization,   
System.Buffers;

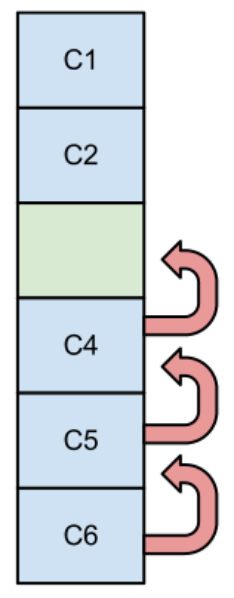
**Trabalhando com Parâmetros:** de entrada, de saida, objeto parâmetro, funções como pârametros, refletindo sobre efeitos colaterais (Imutabilidade e porque é importante).

<https://pt.stackoverflow.com/questions/15510/o-que-imut%C3%A1vel-realmente-significa>

O que são e para que servem Interfaces, e o que são Generics?

**Coleções 01 (System.Collections)** –   
Arrays~~,~~Dictionary~~,~~SortedDictionary~~,~~HashSet~~,~~Listas,   
SortedList,   
Queues,   
Stack.   
**Quais são as diferenças entre esses tipos, tem que métodos distintos, qual o uso indicado?**

Arrays nos possibilitam guardar uma quantidade de elementos e depois acessá-los de forma fácil.  
Mas o problema é que manipular um array não é fácil. Por exemplo, imagine um array com 5 contas guardadas. Se quisermos remover a posição 1, como fazemos? Pois, se apagarmos, precisaremos reordenar todo nosso array. E para inserir um elemento no meio do array? Precisamos "abrir um buraco" no array, empurrando elementos pra baixo, para aí sim colocar o novo elemento no meio.  
Exemplo:



Para resolver os problemas do array, podemos trabalhar com uma classe do C# chamada List. Para utilizarmos uma lista dentro do código precisamos informar qual é o tipo de elemento que a lista armazenará.

Quando precisamos que a operação de busca seja rápida, utilizamos os conjuntos do C# ao invés das listas. Conjuntos são estruturas nas quais podemos fazer buscas rápidas e que não permitem repetição de elementos. Um dos tipos de conjuntos disponíveis no C# é a classe HashSet. Para buscar de maneira rápida, o HashSet "categoriza" os seus elementos, de forma a encontrá-los rapidamente. Por exemplo, imagine você em um supermercado. Se você quer comprar sorvete, você não olha todos os itens do supermercado, mas sim vai direto para a seção de congelados. Lá, você procura o seu sorvete favorito. Veja que você olhou muito menos elementos, pois foi direto para a categoria dele. O HashSet faz a mesma coisa. Ele dá "categorias" para cada um dos elementos, e quando busca por eles, vai direto para a categoria.

Em muitas aplicações além da busca rápida, também precisamos manter a ordenação dos elementos de um conjunto. Nesse tipo de aplicação, podemos utilizar uma nova classe do C# chamada SortedSet.

A diferença é que no HashSet os elementos são espalhados em categorias e por isso não sabemos qual é a ordem da iteração, já o SortedSet guarda os elementos na ordem crescente.

As listas e os conjuntos são duas estruturas que expõem muitos métodos em comum, mas que também possuem diversas diferenças:

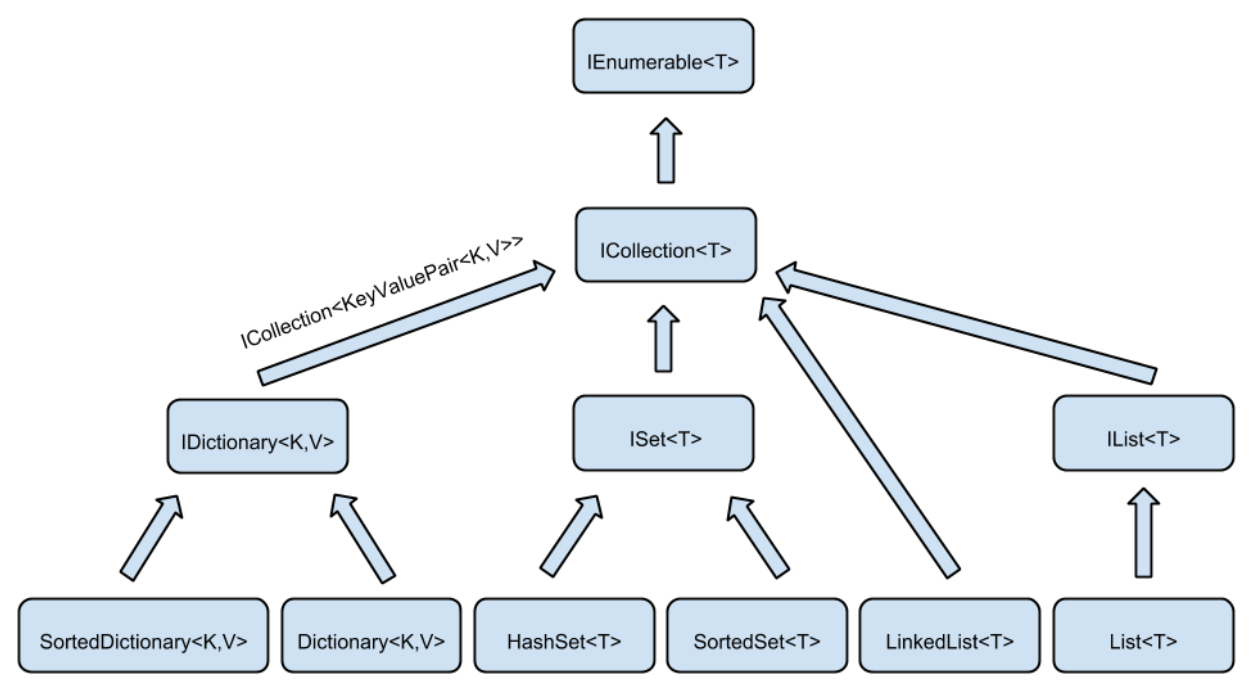
* Nas listas os elementos são armazenados na ordem de inserção enquanto cada conjunto armazena os elementos na ordem que desejar para otimizar o tempo de busca;
* Listas aceitam repetições enquanto os conjuntos não;
* Podemos acessar elementos de uma lista através de seu índice, uma operação que não faz sentido no conjunto.

Como listas e conjuntos possuem muitas operações em comum, tanto as listas quanto os conjuntos implementam uma outra interface do C# chamada ICollection.

Além disso, podemos utilizar o foreach com qualquer coleção do C#. Isso acontece porque o foreach aceita qualquer classe que implemente a interface IEnumerable, que é uma super interface (interface pai) da ICollection.

O Dictionary é uma classe que consegue associar uma chave a um valor. Utilizando o dicionário, podemos, por exemplo, associar o nome do titular com uma conta do sistema. Quando vamos construir um dicionário dentro do código, precisamos informar qual é o tipo da chave e qual será o tipo do valor associado a essa chave, para implementarmos a busca de contas, precisaríamos de um dicionário que associa uma chave do tipo string com uma Conta.

A hierarquia das coleções do C# fica da seguinte forma:





Referencias:

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

<https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers>

<https://www.caelum.com.br/apostila-csharp-orientacao-objetos/>

<https://www.devmedia.com.br/principais-conceitos-da-programacao-orientada-a-objetos/32285>

<https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/structs>

<https://docs.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp/structs>

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

<https://www.tutlane.com/tutorial/csharp/csharp-value-type-and-reference-type-with-examples>

<https://www.geeksforgeeks.org/c-sharp-constructors/>

<http://www.macoratti.net/18/09/c_constr1.htm>

<https://www.geeksforgeeks.org/c-sharp-constructors/>

<http://www.macoratti.net/12/12/c_col1.htm>