04

SortedList

Transcrição

Trabalharemos com um novo projeto *console application,* e demonstraremos uma nova coleção do .NET Framework.

Começaremos trabalhando com o dicionário, como vimos na Parte 1 do curso de C# Collection. (https://cursos.alura.com.br/course/csharp-collections).

Criaremos um dicionário de alunos com quatro nomes diferentes, Vanessa, Ana, Rafael e Wanderson.

O código do aluno será uma string, com duas letras, e em seguida, teremos o nome do aluno.

```
namespace A1._1_SortedList
{
    class Program
    {
        static void Main(string[]  args)
            {
        //vamos criar um dicionário de alunos
        //VT, Vanessa, 34672
        //AL, Ana, 5617
        //RN, Rafael, 17654
        //WM, Wanderson, 11287
```

Trabalharemos com uma classe que estudamos na Parte 1, que é a "Aluno.cs". Ela tem o nome do aluno, número da matrícula, e um construtor que recebe o nome e matrícula, para preencher os dados do aluno.

```
private string nome;
public string Nome
{
                get  {  return nome;  }
}

private int numeroMatricula;
public int NumeroMatricula
{
                get  {  return  numeroMatricula;   }
}
```

Retornaremos ao "Program.cs" e faremos a implementação do dicionário. Declararemos uma interface `iDictionary<>` e, entre `<>` definiremos o tipo da chave e o tipo do valor deste dicionário.

O tipo da chave é uma string, e, como queremos armazenar alunos, o tipo da chave será `Aluno` .

O dicionário se chamará "alunos", e será igual a uma nova instância de um `Dictionary` .

```
IDictionary<string, Aluno> alunos
            = new Dictionary<string, Aluno>();
```

Adicionaremos os alunos, com a função `Add` e, nos parâmetros, inseriremos a chave e o aluno, que no caso são as duas letras, e o nome com matrícula, respectivamente.

```
IDictionary<string, Aluno> alunos
            = new Dictionary<string, Aluno>();

alunos.Add("VT", new Aluno("Vanessa", 34672));
alunos.Add("AL", new Aluno("Ana", 5617));
alunos.Add("RN", new Aluno("Rafael", 17645));
alunos.Add("WM", new Aluno("Wanderson", 11287));
```

Faremos com que as informações sejam impressas no console. Para cada `aluno` da coleção `alunos` utilizaremos a função `Console.WriteLine()`.

```
//imprimindo...
foreach (var aluni in alunos)
{
                Console.WriteLine(aluno);
}
```

Executaremos o programa utilizando o atalho "Ctrl + F5".

Surgirá uma caixa de diálogo em que estarão escritos os nomes e dados dos alunos, sem nenhuma ordem em particular. No caso, ele simplesmente imprimiu na ordem em que as informações foram inseridas.

Para colocarmos isso em teste, removeremos o aluno "Ana", e adicionaremos o aluno "Marcelo".

```
//vamos remover: AL
alunos.Remove("AL");
//vamos.adicionar: MO
alunos.Add("MO", new Aluno("Marcelo", 12345));
```

Imprimiremos novamente, com o mesmo código utilizado acima.

```
//vamos remover: AL
alunos.Remove("AL");
//vamos.adicionar: MO
alunos.Add("MO", new Aluno("Marcelo", 12345));
Console.WriteLine();
foreach (var aluni in alunos)
{
                Console.WriteLine(aluno);
}
```

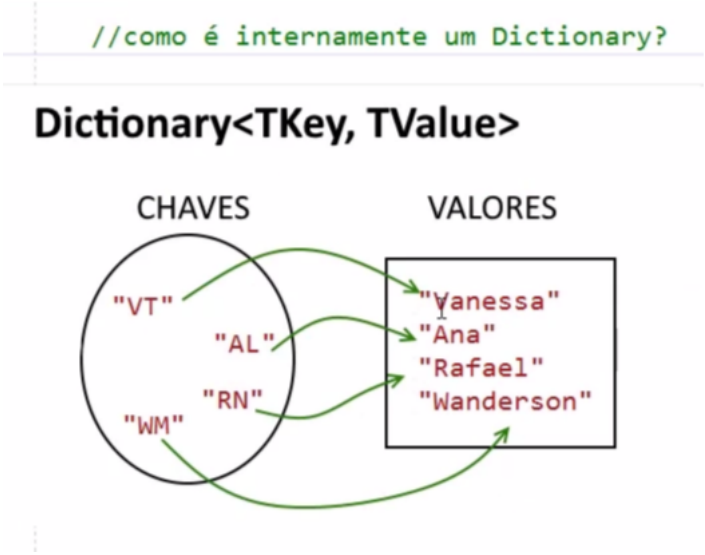
Executaremos novamente o programa.

Veremos que foram escritos os mesmos nomes na primeira impressão mas, na segunda, o nome "Ana" deu lugar a "Marcelo".

O novo nome não foi inserido ao final da lista, como poderia ser esperado.

Caracteristicamente, não é possível sabermos com precisão a localização dos itens no momento da impressão.

Veremos agora o funcionamento interno de um dicionário, por meio de um diagrama.



As chaves, que estão na primeira coleção, são armazenadas de uma forma não-ordenada. Isso significa que, a posição que cada chave ocupa na memória, independe da sua ordem de inserção.

A ordem dependerá do seu código de espalhamento (ou "*hash*"), que é calculado para direcionar um item para determinado lugar na memória, um *bucket*, ou uma gaveta, onde cada informação é organizada, para ser recuperada posteriormente.

O `Dictionary` , internamente, utiliza um *hash* para armazenar as chaves.

Do outro lado, temos os valores correspondentes a cada uma das chaves, sem nenhuma especificação de ordem. Por exemplo, não foram organizadas alfabeticamente.

Para as organizarmos, deveremos utilizar um outro tipo de coleção.

Chamaremos a nova classe de `SortedList` , sendo que os alunos serão os mesmos do exemplo anterior.

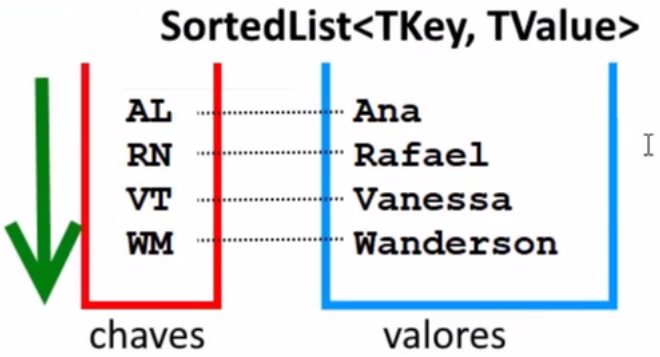
Imprimiremos a coleção, utilizando um laço `foreach` .

```
//apresentando nova coleção...SortedList

IDictionary<string, Aluno>  sorted
        = new SortedList<string, Aluno>();

sorted.Add("VT", new Aluno("Vanessa", 34672));
sorted.Add("AL", new Aluno("Ana", 5617));
sorted.Add("RN", new Aluno("Rafael", 17645));
sorted.Add("WM", new Aluno("Wanderson", 11287));

Console.WriteLine();
foreach (var item in collection)
{
            Console.WriteLine(item);
}
```

Executaremos o programa, utilizando o atalho "Ctrl + F5".

Veremos em primeiro lugar, a lista como a inserimos no primeiro exemplo, em seguida, a lista com a substituição da aluna "Ana" e, por último, os nomes dos alunos do primeiro exemplo, listados em ordem alfabética.

Em outro diagrama, entenderemos como isso funciona internamente.



Temos uma estrutura diferente de armazenamento de chaves e valores. Não teremos a coleção que utiliza o "hash", em vez disso, haverá uma lista automaticamente ordenada. Ou seja, cada vez que inserimos um valor, ele será ordenado automaticamente.

Em outra lista, serão armazenados os valores.

Com isso, vemos que há uma clara diferença estrutural entre uma `SortedList` e um `Dictionary`.

Lembrando que um `SortedList` não é uma lista, este nome se deve ao fato de que a sua coleção de chaves é armazenada numa lista.

Com o atalho "Alt + F12" abriremos a implementação. Com isso, veremos que o `SortedList` implementa o `IDictionary`.

Abaixo, temos a coleção de chaves,"Keys", que tem como retorno um tipo `IList` do tipo `TKey`. Acima, podemos observar a lista de valores `IList<TValue> Values`.

Ambas são representadas pelo diagrama.

Assim, vemos como funciona um "SortedList" e como pode ser utilizado para ordenar automaticamente um dicionário, pelas suas chaves.