



PROGRAMAÇÃO I

AULA 2



Prof. Alan Matheus Pinheiro Araya

CONVERSA INICIAL

O objetivo desta aula é abordar os principais conceitos e tipos de dados utilizados para lidar com Arrays em C#. Vamos explorar as diversas coleções, suas heranças e formas mais comuns de uso para o dia a dia.

TEMA 1 – TIPOS E INTERAÇÕES BÁSICAS COM ARRAYS

No C#, podemos encontrar o tipo básico “Array”, como uma implementação nativa da CLR. Porém, ele pode acabar sendo um tipo muito básico e demandar demasiado esforço quando você precisa implementar controles mais avançados sobre listas de dados. No C#, você tem à sua disposição uma série de classes para lidar com Arrays de forma mais especializada. Essas classes são chamadas de “*Collections*” (Coleções), e oferecem implementações para os mais variados cenários, como listas, filas, pilhas, listas encadeadas, hashsets, entre outros.

No C#, podemos construir um Array básico (nativo da linguagem) simplesmente escrevendo “*type[]*”, onde “*type*” é a *class/valuetype* a partir da qual desejamos criar uma lista. Veja o exemplo de um Array de inteiros, caracteres e outras classes:

```
//Iniciando um array nativo
int[] inteiros = new int[10]; //um array de inteiros contendo 10 posições
char[] caracteres = new char[2] { 'o', 'i' }; //um array de caracteres com
2 posições já inicializadas na construção

var versoes = new Version[5]; //um array de uma classe qualquer
var matriz = new int[3,2]; //uma matriz 3x2 (array bi-dimensional)
var matriz3D = new int[3,3,3]; //uma matriz 3x3x3 (array tri-dimensional)
```

Todo Array no C# herda da classe `System.Array`. Ela é a classe base para Arrays de uma ou mais dimensões (como no exemplo). A classe Array unifica os *types*, assim todas as coleções podem compartilhar métodos e comportamentos previstos no `System.Array`. Mesmo o `System.Array` implementa algumas **Interfaces** (comportamentos) do “universo” de “*Collections*”. Veremos que essas coleções incluem desde tipos simples de Arrays até tipos complexos e específicos. Veja, no quadro, os *namespaces* das principais coleções do C#:



Quadro 1 – Namespace: coleções do C#

Namespace	Recursos/Classes disponibilizados
System.Collections	Classes e interfaces de coleções não genéricas (IEnumerable, IEnumerator etc.)
System.Collections.Specialized	Classes e coleções não genéricas especializadas (StringDictionary, OrderedDictionary, NameValueCollection etc.)
System.Collections.Generic	Principais classes e interfaces de coleções genéricas (List<T>, IEnumerable<T>, Dictionary<T> etc.)
System.Collections.ObjectModel	Classes base para tipos customizados de coleções
System.Collections.Concurrent	Coleções do tipo “Thread Safe”; podem ser utilizadas em cenários multi-thread

Fonte: Albahari; Albahari, 2017, p. 301

Antes de abordar as coleções especializadas, é preciso **dominar** um tema muito importante: “**Generics**” (Tipos genéricos) do C#.

TEMA 2 – GENERICS NO C#

O C# tem dois mecanismos separados para a escrita de código, que podem ser reutilizados em diferentes tipos: **herança (inheritance)** e **genéricos (generics)**. Enquanto a herança expressa a capacidade de reutilização de um “*Base Type*”, os genéricos expressam capacidade de reutilização com um “modelo” (*template*). Os *Generics*, quando comparados à herança, podem aumentar a segurança de conversões entre Types e reduzir a carga de “boxing, unboxing e casting” entre Types de um mesmo modelo (Albahari; Albahari, 2017, p. 122).

Os Generics resolvem um problema muito comum em coleções, que é a generalização da coleção para trabalhar com qualquer Type (tipo de objeto). Para declarar uma classe que faça uso desse recurso, basta colocar entre “<>”, por exemplo: `MinhaClasse<T>`. Dentro do sinal “<” e “>”, poderá ser usada qualquer palavra (seguindo as regras de nomeação de classes), mas comumente você encontra letras, em especial a letra “T”, que é um acrônimo para Type.



A seguir, veja um exemplo de uma classe do tipo “Pilha” (*Stack*), que trabalha com qualquer Type, fazendo uso de Generics para facilitar o processo de manipulação dos Types:

```
/// <summary>
/// Classe customizada de uma Stack simples.
/// Observe que ela declara um template onde aceita qualquer Type como pa
râmetro dentro do <>
/// </summary>
public class CustomStack<T>
{
    //posição atual do ponteiro da pilha
    int position;

    //Quando declaramos que nossa classe irá trabalhar com Generics,
    usando a notação <T>
    //podemos passar qualquer Type/objeto para o T
    //E podemos também usar o T como um "placeholder", ou seja,
    //um acrônimo dentro de nossa classe, para referência um Type qualquer
    //Veja, nesse exemplo, não sabemos o que é o "T"
    //Apenas sabemos que será um Type qualquer (qualquer tipo derivado de
    object)
    T[] arrayDoTipoT = new T[100];

    public void Push(T obj)
    {
        arrayDoTipoT[position++] = obj;
    }

    public T Pop()
    {
        return arrayDoTipoT[--position];
    }
}

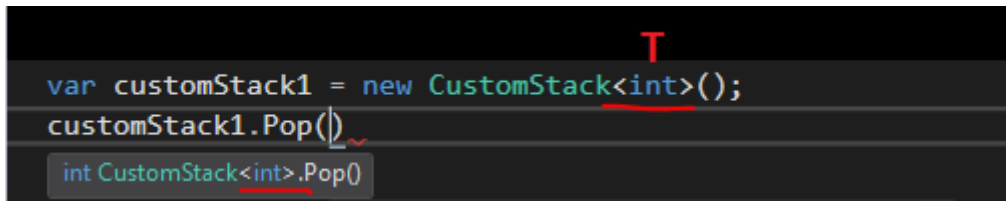
public class ExemploStack
{
    public ExemploStack()
    {
        var customStackInstance = new CustomStack<string>();
        customStackInstance.Push("oi pessoal!");
        customStackInstance.Push("Essa é uma fila!");
        customStackInstance.Push("É uma estrutura de dados que trabalha co
m LIFO.");
        customStackInstance.Push("LIFO-> Last in, First Out.");
        customStackInstance.Push("fim...");

        string popResult = customStackInstance.Pop();
        popResult += customStackInstance.Pop();
        popResult += customStackInstance.Pop();
        popResult += customStackInstance.Pop();
        popResult += customStackInstance.Pop();

        Console.WriteLine(popResult);
    }
}
```



O que podemos concluir do exemplo acima é que o Array `T[]` aceitou uma string dentro dele, mesmo sem a declaração de que se tratava de um array de strings na declaração da classe `CustomStack`. Definimos o Type do `T` no momento em que inicializamos a nossa `Stack`, com a instrução: `new CustomStack<string>()`. Nesse momento, estamos falando para o compilador: “Nossa `Stack` será genérica para qualquer `String`”. Veja na imagem a seguir como a IDE do Visual Studio já entende o tipo e indica o tipo correto a ser fornecido após a inicialização:

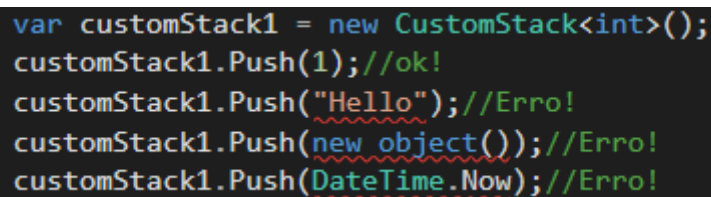


```
var customStack1 = new CustomStack<int>();  
customStack1.Pop();  
int CustomStack<int>.Pop()
```

Nesse momento, o C# entende que a instância “`customStack1`” trabalhará com inteiros. Veja no exemplo a seguir que podemos passar qualquer objeto em instâncias diferentes:

```
var customStack1 = new CustomStack<int>();  
var customStack2 = new CustomStack<string>();  
var customStack3 = new CustomStack<DateTime>();  
var customStack4 = new CustomStack<object>();  
var customStack5 = new CustomStack<Enum>();
```

Um erro comum dos iniciantes em Generics é interpretá-la equivocadamente com um tipo dinâmico, em que você pode passar vários Types para uma mesma instância que suporte Generics e que já tenha o seu Type definido na inicialização. Nesse caso, ocorrerá um erro de compilação, como podemos ver a seguir:



```
var customStack1 = new CustomStack<int>();  
customStack1.Push(1); //ok!  
customStack1.Push("Hello"); //Erro!  
customStack1.Push(new object()); //Erro!  
customStack1.Push(DateTime.Now); //Erro!
```

Após definir o Type que será utilizado na instância de uma classe que usa *Generics*, seria equivalente a ela substituir o valor entre `<>` pelo Type, como no exemplo a seguir (repare no **negrito** utilizado para simbolizar a **hipotética** substituição do `T` pelo type “`string`”). Note que essa substituição ocorrerá **apenas** no momento da **inicialização** da nossa classe `CustomStack`:



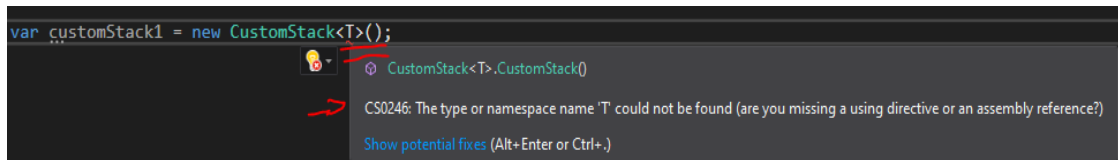
```
public class CustomStack<string>
{
    int position;

    string[] arrayDoTipoT = new string[100];

    public void Push(T obj)
    {
        arrayDoTipoT[position++] = obj;
    }

    public string Pop()
    {
        return arrayDoTipoT[--position];
    }
}
```

Tecnicamente, dizemos que a *CustomStack<T>* é um “tipo aberto” (*open Type*), enquanto a *CustomStack<int>* é um “tipo fechado” (*closed Type*). Em tempo de execução (*runtime*), todas as instâncias de classes que usam *Generics* são consideradas “tipos fechados” (*closed types*), com seus modelos (*placeholders*) preenchidos. Isso nos leva à conclusão de que tentar inicializar um *Generics* sem definir o seu *Type* é impossível, pois levará a um erro (Albahari; Albahari, 2017, p. 123).



2.1 Por que os *Generics* existem? *Object[]* não resolve?

Os *Generics* existem para escrever código reutilizável entre diferentes tipos. Suponha que queremos usar uma pilha (*Stack*) de inteiros, mas não existem os tipos genéricos. Uma solução seria codificar uma classe separada para cada *Type* necessário, por exemplo: *IntStack*, *StringStack* etc. Claramente, isso causaria uma **duplicação** de código considerável.

Outra solução seria escrever uma pilha que é genérica usando o *System.Object* como base, certo? Afinal de contas, todos os *Types* são herdeiros dele, e assim poderíamos ter uma classe *StackObject* capaz de trabalhar com qualquer objeto sem o uso dos *Generics*. Algo como (Albahari; Albahari, 2017, p. 124):



```
public class ObjectStack
{
    int position;
    object[] data = new object[10];
    public void Push(object obj)
    {
        data[position++] = obj;
    }
    public object Pop()
    {
        return data[--position];
    }
}
```

Apesar dessa solução funcionar, em termos de compilação, ela não é prática e nem performática. Afinal, cada vez que desejarmos acessar um dado dessa Pilha, teremos que referenciar o Type. Esse processo pode inclusive levar a erros em momento de execução do código! Tais erros dificilmente poderiam ser previstos em tempo de compilação (Albahari; Albahari, 2017, p. 124):

```
ObjectStack stack = new ObjectStack();
stack.Push("uma string qualquer"); // Armazenamos uma string -- boxing
stack.Push(100); // Depois um int -- boxing

// Para converter o tipo para int, precisamos fazer um unboxing explícito
// (explicit cast) - Nesse caso vamos gerar um erro Erro!
int i = (int)stack.Pop();

string s = (string)stack.Pop();// Outro Erro!
```

Além de induzir a erros em tempo de execução, o código acima ainda terá uma performance muito mais baixa que a utilização de *Generics*, pois precisará executar processos de *boxing* e *unboxing* sempre que adicionarmos ou removermos (respectivamente) um item da pilha (*Stack*). O mesmo aconteceria se inicializássemos a nossa classe *CustomStack* utilizando a notação “*CustomStack<object>*”.

2.2 Restrições de tipos utilizando Generics

Podemos utilizar um conceito chamado “Generics Constraints” para limitar ou restringir o tipo que desejamos em nossas classes. As notações possíveis são (Albahari; Albahari, 2017, p. 127):

- class MinhaClasse where T : *OutraClasseQualquer* – Indica que o *Type* passado no T deve **herdar** da classe “*OutraClasseQualquer*”.



- class MinhaClasse where T : *interface* – Indica que o *Type* passado no T deve **implementar** a **interface** especificada.
- class MinhaClasse where T : *class* – Indica que o *Type* passado no T **deve ser** do tipo “**Reference Type**”, ou seja, não pode ser um “*Value Type*”, como *structs*, *int*, *char* etc.
- class MinhaClasse where T : *struct* – Indica que o *Type* passado no T **deve ser** do tipo “**Value Type**”, ou seja, não pode ser um “*Reference Type*”.
- class MinhaClasse where T : *new()* – Indica que o *Type* passado no T **deve apresentar** um construtor sem parâmetros.

No exemplo a seguir, podemos ver a utilização de uma classe “GenericClass<T,U>”, que requer que o T seja derivado (ou idêntico a) “Classe1”, implementando a interface “Interface1”. Também requer que o tipo U provenha de uma classe sem parâmetros no construtor (Albahari; Albahari, 2017, p. 127):

```
public class Classe1 { }
public interface Interface1 { }

public class GenericClass<T, U> where T : Classe1, Interface1
                                where U : new()
{
}
```

Observe que, no exemplo, também trabalhamos com a possibilidade de mais de um parâmetro genérico. Isso é perfeitamente possível. Uma classe pode ter quantos parâmetros genéricos desejarmos.

No C#, é possível implementar restrições de parâmetros *Generics* também nos métodos. A notação é igual à declaração nas classes. No exemplo a seguir, temos uma classe com *Generics* e uma restrição para o parâmetro genérico T herdar/ser uma *struct*. Temos ainda um método chamado Max<T>, que retorna o maior T entre dois parâmetros T passados. Porém, existe uma **restrição** para chamar esse método. Além de herdar/ser uma *struct* do tipo T, é necessário que o T desse método implemente a interface “IComparable”, que provê um método que compara dois tipos, retornando 0 se forem iguais, 1 se o segundo parâmetro for maior que o primeiro, e -1 se o primeiro for maior que o segundo. Note que o tipo “int” implementa essa interface.



```
public class Classe2<T> where T : struct
{
    public T Max<T>(T param1, T param2) where T : IComparable
    {
        return param1.CompareTo(param2) > 0 ? param1 : param2;
    }
}
```

```
> var teste = new Classe2<int>();
. teste.Max<int>(5, 10)
10
```

Observamos na imagem o resultado da execução do método `Max<int>`, recebendo no *param1* o valor 5 e no *param2* o valor 10.

Também podemos observar, no exemplo, que foi utilizada uma notação de “if”, chamada de “if ternário”, que simplifica o “if” em uma única linha. Esse tipo de notação deve ser usado com cautela para não prejudicar a leitura do código.

Saiba mais

Para saber mais sobre if ternário, consulte a documentação oficial no link:

MICROSOFT. **Operador?: (referência do C#)**. 17 set. 2020. Disponível em: <<https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/conditional-operator>>. Acesso em: 3 set. 2021.

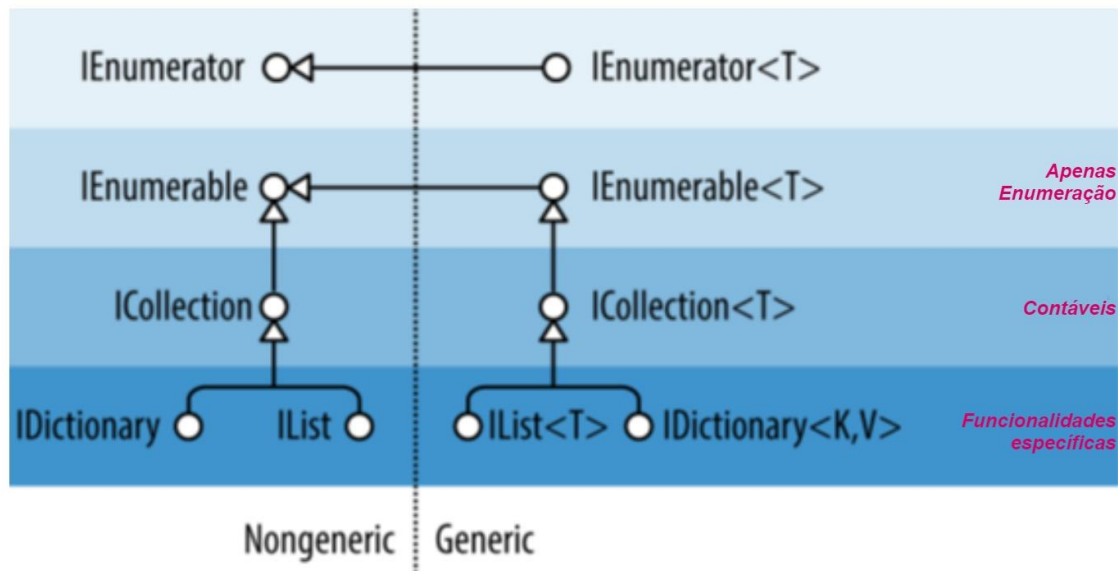
TEMA 3 – ARRAYS, LISTS, SETS, DICTIONARYS E COLLECTIONS

Como vimos no Tema 1, o C# apresenta tipos específicos e mais complexos para coleções e listas de dados. Quando pensamos em coleções, as diferenciamos de simples *arrays*, pois apresentam propriedades de armazenamento não necessariamente lineares. Porém, todas têm uma característica comum: a possibilidade de percorrer os seus índices transversalmente (elemento a elemento). Pensando nisso, o C# oferece um par de **Interfaces** (comportamentos comuns) Genéricos e Não-Genéricos para que você possa percorrer facilmente as coleções, sem se preocupar com as estruturas complexas. Essas interfaces, por sua vez, apresentam um conjunto de outras interfaces que as herdam, formando assim uma árvore complexa de



comportamentos que são comuns a determinados **tipos** de coleções. Veja a figura a seguir.

Figura 1 – Interfaces: IEnumerable e IEnumerator



Fonte: Albahari, 2017, p. 302.

Na imagem, podemos ver as duas principais interfaces *IEnumerator* e *IEnumerator*, com seus pares generics. Observe que a interface *IEnumerator* permite apenas enumerar (percorrer) elementos, enquanto a interface *ICollection* permite, além de enumerar, contar os objetos e manter o controle dessa quantidade. As interfaces *IList* e *IDictionary* apresentam muito mais funcionalidades específicas (Albahari; Albahari, 2017, p. 302).

3.1 IEnumerable e IEnumerator

A interface *IEnumerator* define o protocolo (comportamento) **básico de baixo nível**, pelo qual os elementos em uma coleção são percorridos (ou enumerados) apenas de maneira sequencial. Sua declaração é a seguinte (Albahari; Albahari, 2017, p. 302):

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```



O método *MoveNext* avança o elemento atual ou “cursor” para a próxima posição, retornando *false* se não houver mais elementos na coleção. A propriedade (*read-only*) *Current* retorna o elemento na posição atual. O método *MoveNext* deve ser chamado antes de recuperar o primeiro elemento – isso é necessário para permitir coleções vazias. O método *Reset*, se implementado, retorna ao início, permitindo que a coleção seja enumerada novamente. O *Reset* existe principalmente para interoperabilidade COM. Chamá-lo diretamente é em geral evitado, porque ele não é universalmente suportado, sendo desnecessário, pois é mais fácil instanciar um novo enumerador (Albahari; Albahari, 2017, p. 302).

Coleções geralmente vão implementar a interface **IEnumerable**, possivelmente a interface **mais utilizada de todo o C#**:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Definindo um único método e comportamento, essa interface nos retorna um *IEnumerator*. Com isso, temos grande flexibilidade, pois em primeiro lugar significa que vários trechos de código podem enumerar (percorrer) a coleção ao mesmo tempo (**uma vez que, para cada *for* ou *foreach*, o C# irá instanciar um enumerador diferente e lhe retornar – esse, por sua vez, permitirá que você percorra a coleção sem interferir nos demais).**

Um exemplo de código usando apenas a interface base “*IEnumerator*” pode ser implementado para percorrer os caracteres de uma string:

```
string helloString = "Hello";

// Uma vez que a classe string implementa a interface IEnumerable, podemos
// chamar o método GetEnumerator():
IEnumerator enumerator = helloString.GetEnumerator();
while (enumerator.MoveNext())
{
    char c = (char)enumerator.Current;
    Console.Write(c + ".");
}

// Resultado no console : H.e.l.l.o.
```



Porém, é muito raro ter de chamar os métodos do *IEnumerator* de forma direta. O C# provê um atalho muito mais simples: **foreach**. O bloco (e palavra reservada) **foreach** automaticamente instancia o enumerador e o percorre executando o *MoveNext*. Veja o mesmo exemplo usando o *foreach*:

```
string helloString = "Hello";  
  
foreach (char c in helloString)  
{  
    Console.Write(c + ".");  
}  
// Resultado no console : H.e.l.l.o.
```

Porém como vimos anteriormente, em Generics usar a interface *IEnumerable* “pura” não é nada conveniente. O retorno do método *MoveNext* é um objeto. Isso gera problemas de boxing e unboxing. Por isso, você vai notar que quase todas as coleções implementam a interface *IEnumerable<T>* (com generics), provendo assim uma forma segura de acesso ao Type correto do objeto da coleção (Griffiths, 2019, p. 246).

3.2 *ICollection<T>*

Embora as interfaces de *IEnumerable* forneçam um comportamento para iteração “somente para frente” sobre uma coleção, elas não fornecem um mecanismo para determinar o tamanho da coleção, acesso a um membro por meio índice, pesquisa ou modificação da coleção. Para tal funcionalidade, o C# define as seguintes interfaces: ***ICollection***, ***IList*** e ***IDictionary***. Cada uma vem em versões genéricas e não genéricas. As versões não genéricas existem principalmente para suporte de legado a versões mais antigas do framework (Albahari; Albahari, 2017, p. 309).

Podemos resumir as funcionalidades das principais interfaces de enumeração, coleção e listas, com os seguintes pontos:

- ***IEnumerable<T>* (e *IEnumerator*)**
 - Prove funcionalidades básicas e simples (apenas enumeração)
- ***ICollection<T>* (e *ICollection*)**
 - Prove funcionalidades mais avançadas de contagem e acesso aos elementos da coleção
- ***IList <T>* e *IDictionary <K,V>***



- Prove o maior número de funcionalidades, sendo as interfaces mais comumente utilizadas (provem indexação, métodos de busca etc.)

A interface `ICollection <T>` é a interface padrão para coleções contáveis (que apresentam a propriedade *Count*) de objetos. Há a capacidade de determinar o tamanho de uma coleção (contagem), determinar se existe um item na coleção (método *Contains*), copiar a coleção para um Array (método *ToArray*), e ainda determinar se a coleção é somente leitura (propriedade *IsReadOnly*). Há ainda métodos para Adicionar (*Add*), Remover (*Remove*) e Limpar (*Clear*) itens da coleção. Como ela também herda de `IEnumerable <T>`, pode ser percorrida (enumerada) por meio da instrução *foreach* (Albahari; Albahari, 2017, p. 310):

```
// Interface ICollection no C#  
  
public interface ICollection<T> : IEnumerable<T>, IEnumerable  
{  
    int Count { get; }  
    bool Contains(T item);  
    void CopyTo(T[] array, int arrayIndex);  
    bool IsReadOnly { get; }  
  
    void Add(T item);  
    bool Remove(T item);  
    void Clear();  
}
```

O C# provê uma classe concreta que implementa a interface, a classe `Collection<T>`. Ela irá prover os mesmos métodos disponíveis na interface e ainda outros métodos existentes em outras interfaces (como o método `IndexOf`). Isso acontece porque a classe concreta `Collection<T>` implementa uma série de interfaces, inclusive `ICollection<T>`. Afinal, o uso real de uma coleção acaba necessitando de comportamentos disponíveis apenas na interface `ICollection`. Veremos a seguir que essa é a interface de coleções mais completa do C#.

3.3 `ICollection<T>`

A `ICollection <T>` é a interface padrão para coleções indexáveis, o que significa que você pode acessar um elemento especificando diretamente o seu índice. Ela também herda todas as funcionalidades das interfaces `ICollection<T>` e `IEnumerable<T>`, fornecendo a capacidade de ler ou escrever um elemento de



uma posição específica (por meio de um indexador) e inserir/remover um elemento por posição (Albahari; Albahari, 2017, p. 311):

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    T this[int index] { get; set; }
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);
}
```

O método *IndexOf* realiza uma busca linear, retornando o index do elemento encontrado, ou -1 quando não encontra o elemento desejado.

Como podemos perceber, a interface *IList<T>* provê um nível de funcionalidade ideal para trabalhar com coleções e listas de objetos. Em sua implementação concreta, a classe *List<T>* será uma das classes mais utilizadas durante a codificação em C#. Vamos ver a seguir alguns exemplos de funcionamento da classe *List<T>* que são comuns em nosso dia a dia:

```
//Exemplo 1:
//declaração de uma variável IList com sua inicialização pelo tipo concreto
IList<int> lista1 = new List<int>();

lista1.Add(1);
lista1.Add(1000000);

Console.WriteLine("Count da lista1: " + lista1.Count);
//Output no Console: Count da lista1: 2

//verificamos se a lista contém um elemento
if (lista1.Contains(1000000))
{
    //removemos o elemento
    lista1.Remove(1000000);
}

//Verificamos a quantidade presente na lista
Console.WriteLine("Count da lista1: " + lista1.Count);
//Output no Console: Count da lista1: 1
```



```
//Exemplo2:
//declaração de variável concreta com quantidade inicial alocada
List<string> lista2 = new List<string>(10);

lista2.Add("olá");
lista2.Add("mundo!");

Console.WriteLine("Count da lista2: " + lista2.Count);
Console.WriteLine("Capacidade da lista2: " + lista2.Capacity);
//Output no Console:
//Count da lista2: 2
//Capacidade da lista2: 10

lista2.AddRange(new string[] { "exemplo","de multiplos","valores","em um
a lista","ao mesmo","tempo!"," ","","Observe",
    "que estamos","passando", "multiplas strings em um array
","podemos fazer isso","pois o método AddRange",
    "recebe um IEnumerable<T>"}));

//Agora observe que inicializamos a Lista com o valor 10 no construtor
//isso significa que ela começou alocando 10 espaços na memória
//mas ao realizar o AddRange, vamos passar desse limite
//a lista irá se auto-dimensionar sozinha para comportar o novo tamanho

Console.WriteLine("Count da lista2: " + lista2.Count);
Console.WriteLine("Capacidade da lista2: " + lista2.Capacity);
//Output no Console:
//Count da lista2: 16
//Capacidade da lista2: 20

//Abaixo vamos fazer um foreach nos elementos e
//escreve-los no console, separados por um espaço
foreach (var elemento in lista2)
{
    Console.Write(elemento + " ");
}
//Output no console:
//olá mundo! exemplo de multiplos valores em uma lista ao mesmo tempo!
//Observe que estamos passando multiplas strings em um array, podemos faz
//er isso, pois o método AddRange recebe um IEnumerable<T>
```



```
//Exemplo 3
//Inicializando a lista já com conteúdo alocado
var lista3 = new List<string>() { "elemento1", "elemento2", "elemento3",
    "elemento4", "elemento5" };

//O método Insert insere um elemento em uma posição da lista, deslocando
//o index dos demais a sua direita
lista3.Insert(2, "elemento6");

//Escrevemos no console o index do elemento4
//ele começou com o index 3 e agora estará com o index 4
Console.WriteLine(lista3.IndexOf("elemento3"));

//Remove um elemento de um index específico
lista3.RemoveAt(1);

//Inverte a ordem dos indices de uma lista
lista3.Reverse();

foreach (var elemento in lista3)
{
    Console.Write(elemento + ", ");
}

//Output no console:
//elemento5, elemento4, elemento3, elemento6, elemento1,
```

```
//Exemplo 4
var lista4 = new List<string>() { "elemento1", "elemento2", "elemento3"
};

//Aqui estamos usando a propriedade de indexador da lista para buscar um
//elemento em uma posição específica do array
//passando o index do elemento a lista realiza uma busca "ótima" e nos r
//etorna rapidamente o elemento
var elementoEncontrado = lista4[2];

Console.WriteLine("elementoEncontrado: " + elementoEncontrado);

//Vamos popular a lista com mais 3 milhões de registros
for (int i = 4; i < 3000000; i++)
{
    lista4.Add("elemento" + i);
}

//Agora vamos usar dois meios de busca:
//1:
//busca pelo conteúdo, usando o "contains". Nesse caso a lista deve comp
//arar o HashCode dos elementos
//a fim de encontrar um elemento igual na lista, fazendo isso linearment
//e (1 por 1) até encontrar o primeiro
//vamos mensurar também o tempo que levamos para encontrar o elemento de
//ssa forma
```




```
var stopwatch4 = Stopwatch.StartNew();

if (lista4.Contains("elemento2500000"))
{
    stopwatch4.Stop();
    Console.WriteLine($"Encontramos o elemento elemento2500000! O seu computador levou: {stopwatch4.ElapsedMilliseconds} milisegundos para encontra-lo!");
}

//2:
//busca pelo index do elemento. Vamos usar o indexador para buscar o mesmo elemento anterior
//nesse caso a lista irá realizar uma busca ótima, retornando "instantaneamente" o elemento
//vamos mensurar também o tempo que levamos para encontrar o elemento dessa forma

stopwatch4.Reset();
stopwatch4.Start();

//como a lista começa em 0, para buscar o elemento 2500000 devemos buscar o index 2500000-1 (2499999)
if (lista4[2500000-1] != null)
{
    stopwatch4.Stop();
    Console.WriteLine($"Encontramos o elemento elemento2500000! O seu computador levou: {stopwatch4.ElapsedMilliseconds} milisegundos para encontra-lo!");
}
```

Nos exemplos, podemos observar alguns dos principais métodos da classe *List<T>* e de sua interface *ICollection<T>*. Métodos como *Add*, *Remove*, *AddRange*, *RemoveAt*, *Contains*, *Reverse*, *RemoveAt* e a busca por um valor usando a propriedade indexadora, isto é, acessando um elemento da lista em uma posição específica do vetor, com a notação “Lista[x]” (o x representa o índice a ser acessado).

Veremos na sequência estruturas de dados mais especializadas que a Lista, com propriedades e comportamentos específicos, mesmo que algumas implementem a interface *ICollection<T>*. Algumas das principais diferenças estão na forma como essas estruturas armazenam os valores em memória. Em nosso caso, pudemos observar que a busca utilizando “*Contains*” (no exemplo 4) leva muito mais tempo do que a busca diretamente pelo índice. Isso porque o computador precisa enumerar os valores, um a um, até encontrar um elemento que contenha um valor de referência igual àquele que passamos. Como a lista não tem um mecanismo de indexação por conteúdo, essa será a sua performance



mais rápida. De outro lado, acessando diretamente o index da lista, ou seja, a posição que aponta diretamente para o espaço em memória onde está o elemento, percebemos que o tempo de retorno da informação é instantâneo! Isso porque, nesse caso, a lista não irá executar um *foreach* nos elementos até encontrá-lo, e sim acessar a sua referência diretamente.

Ainda assim, a lista fornece alguns métodos relacionados à busca de itens, que são fundamentais para as extensões LINQ que veremos mais adiante

- *BinarySearch*: Para pesquisar rapidamente (boa performance) itens previamente ordenados.
- *IndexOf / LastIndex*: Para pesquisar índices de elementos em arrays não ordenados
- *Find / FindLast / FindIndex / FindLastIndex / FindAll / Exists / TrueForAll*: Busca elementos em arrays não ordenados usando um Predicado (*Predicate<T>*). Um Predicado nada mais é que uma função de comparação utilizada para distinguir elementos em buscas.

Saiba mais

Caso deseje se aprofundar mais, consulte a documentação oficial:

MICROSOFT. **Predicate<T>** **Delegar.** Disponível em:
<<https://docs.microsoft.com/pt-br/dotnet/api/system.predicate-1?view=net-5.0>>.
Acesso em: 3 set. 2021.

3.4 HashSet<T> e SortedSet<T>

Os tipos de coleções *HashSet<T>* e *SortedSet<T>* são muito utilizados em cenários de necessidade de performance em buscas. Essas coleções apresentam as seguintes características especiais:

- Seus métodos *Contains* são executados de forma muito rápida usando uma pesquisa baseada em *Hash*.
- Elas não armazenam elementos duplicados e ignoram (se há geração de erros) solicitações para adicionar duplicatas.
- Você não pode acessar um elemento por posição, como faz com a *List*. Isso é uma característica matemática do tipo de estrutura de dados do *HashSet*.



Um `HashSet<T>` é implementado usando o conceito de “*hashtable*”, que armazena apenas chaves. Uma `SortedSet<T>` é implementada com uma árvore que apresenta “*hashtable*” e um índice especial. As duas coleções herdam de `ICollection<T>` e oferecem métodos comuns, que já vimos até agora em coleções, como: *Add*, *Remove* e *Contains*. Para e conhecer mais sobre `HashSets`, você pode Albahari e Albahari (2017, p. 328).

Essas coleções apresentam métodos interessantes para a manipulação de seus elementos. Como elas mantêm os elementos vinculados a uma *hashtable*, algumas operações são destrutivas (recriam a coleção), enquanto outras operações apenas adicionam elementos. As operações **destrutivas** para modificação de Sets são (Albahari; Albahari, 2017, p. 328):

- `public void UnionWith (IEnumerable<T> other);` // Adiciona
- `public void IntersectWith (IEnumerable<T> other);` // Remove
- `public void ExceptWith (IEnumerable<T> other);` // Adiciona
- `public void SymmetricExceptWith (IEnumerable<T> other);` // Remove

As principais operações **não destrutivas** para a modificação de Sets são:

- `public bool IsSubsetOf (IEnumerable<T> other)`
- `public bool IsProperSubsetOf (IEnumerable<T> other)`
- `public bool IsSupersetOf (IEnumerable<T> other)`
- `public bool IsProperSupersetOf (IEnumerable<T> other)`
- `public bool Overlaps (IEnumerable<T> other)`
- `public bool SetEquals (IEnumerable<T> other)`



```
//inicia uma string (que também pode ser interpretada como uma coleção d
e caracteres - char)
var listaChars = "programação 1 - collections";

//inicia um HashSet de caracteres
var letters = new HashSet<char>(listaChars);

//Verifica se a coleção contém alguma letra
Console.WriteLine(letters.Contains('p')); // true
Console.WriteLine(letters.Contains('z')); // false

//Percorremos a coleção escrevendo os caracteres no console
//Observe:
// NÃO há caracteres repetidos
foreach (char c in letters)
{
    Console.Write(c);
}
//Output do Console: progamça 1-cletins
```

Observe no exemplo que o HashSet não armazenou os caracteres duplicados de nossa String. Esse tipo de coleção é muito útil para armazenar dados indexados e garantir uma busca rápida. Vamos utilizá-lo em nossas aulas práticas para exemplificar cenários de uso e performance.

3.5 Dictionary<TKey, TValue>

A coleção Dictionary<TKey, TValue>, assim como a List<T>, é uma das coleções mais úteis e versáteis no C#. Dicionários são coleções muito úteis quando desejamos manter um controle de itens não duplicados através de uma chave, e mesmo assim ter uma boa performance em buscas.

Um Dictionary<TKey, TValue> armazena dentro de si uma estrutura de chave-valor chamada **KeyValuePair**. Essa estrutura armazena os dados na memória usando um conceito de chave (um Type que pode ser referência ou valor e que representa o valor único indexado dentro da Hashtable) e valor (um Type que pode ser por referência, associado à chave). A estrutura KeyValuePair tem a seguinte “anatomia”:

```
public struct KeyValuePair<TKey, TValue>
{
    public TKey Key { get; }
    public TValue Value { get; }
}
```



A coleção de dicionário provê todos os métodos que você já conhece da interface `ICollection<T>`. A principal característica determinante em dicionários é que, ao adicionar itens nos dicionários, há garantia de que o item adicionado seja único, **lançando uma exceção**, quando a chave (key) do item adicionado for duplicada na coleção. Da mesma forma, ao tentar recuperar um item por sua chave, caso o item não exista na coleção, será lançada uma exceção.

Para facilitar o uso, o `Dictionary<TKey, TValue>` provê três métodos muito úteis para o desenvolvedor:

- *ContainsKey* – Retorna um booleano (true/false), indicando se a chave já existe ou não dentro do dicionário.
- *TryGetValue* – Retorna um booleano (true/false), indicando se o elemento existe e foi recuperado com sucesso da coleção, além de retornar o elemento em si via interface de **parâmetros out** do C#.
- *TryAdd* – Tenta adicionar um elemento na coleção. Caso já exista na coleção uma chave conflitante com a nova chave fornecida, retorna false. Caso consiga adicionar com sucesso, retorna true.

Além disso, o dicionário provê duas propriedades para as chaves e para os valores, respectivamente, que são coleções das chaves/valores do dicionário.

No exemplo a seguir, podemos ver operações comuns em um dicionário. Veja a utilização dos métodos *TryGetValue*, *TryAdd* e *ContainsKey*. Também podemos observar a enumeração de elementos através do foreach de três formas diferentes:



```
// Exemplo de dicionário usando uma chave do tipo int
// e um valor do tipo string
var dicionario1 = new Dictionary<int, string>();

//adicionando valores
dicionario1.Add(1, "primeiro valor");
dicionario1.Add(2000, "segundo valor");
dicionario1.Add(50, "terceiro valor");

//atualizando valores pela chave
dicionario1[2000] = "segundo valor - atualizado";

//exemplo chamando o contains
Console.WriteLine(dicionario1.ContainsKey(50)); // true
Console.WriteLine(dicionario1.ContainsKey(4)); // false
Console.WriteLine(dicionario1.ContainsKey(2)); // false

//exemplo do TryAdd de uma chave duplicada (50)
//observe que o retorno é false
Console.WriteLine(dicionario1.TryAdd(50, "valor repetido de chave")); //
false

//exemplo do TryGetValue. Observe a notação de parâmetro out do c#
//o parâmetro out é utilizado para que um método possa retornar
//mais de um valor de sua execução
if (dicionario1.TryGetValue(2000, out string valorOutput))
{
    //o valor da variavel "valorOutput" foi recuperado dentro do TryGetV
    alue
    Console.WriteLine(valorOutput); //segundo valor - atualizado
}

//Exemplo de retorno false do TryGetValue para uma chave inexistente
Console.WriteLine(dicionario1.TryGetValue(300, out string valorOutput2));
//false
Console.WriteLine(dicionario1.TryGetValue(10, out string valorOutput3));
//false

//Exemplo do foreach nos elementos KeyValuePair
foreach (var elementoKeyValuePair in dicionario1)
{
    //Monta string através do principal método ToString()
    Console.WriteLine(elementoKeyValuePair.ToString());

    //Exemplo de acesso Key e Value
    Console.WriteLine(elementoKeyValuePair.Key);
    Console.WriteLine(elementoKeyValuePair.Value);
}

//Exemplo de foreach nas chaves
foreach (var chave in dicionario1.Keys)
{
    Console.WriteLine(chave);
}

//Exemplo de foreach de valores
foreach (var valor in dicionario1.Values)
{
    Console.WriteLine(valor);
}
```



3.6 Stack<T>

O C# ainda provê outras coleções que implementam as mais diversas estruturas clássicas de dados. Como exemplo, temos as Pilhas (*Stack*) e Filas (*Queue*).

A *Stack<T>* é uma estrutura de dados LIFO (último que entra é o primeiro a sair). Seus principais métodos são:

- *Push* – Adiciona um elemento no topo da pilha
- *Pop* – Remove o primeiro elemento do topo da pilha
- *Peek* – Recupera o primeiro elemento do topo da pilha sem removê-lo
- *Count* – Retorna a quantidade de elementos na pilha

A *Stack<T>* implementa internamente um array para as suas coleções. É redimensionado conforme a necessidade de tamanho da pilha, assim como a *Queue<T>* e a *List<T>*. A seguir, veja um exemplo de uso dos principais métodos de uma *Stack*:

```
var stackExemplo = new Stack<int>();
stackExemplo.Push(1); // Pilha = 1
stackExemplo.Push(2); // Pilha = 1,2
stackExemplo.Push(3); // Pilha = 1,2,3

Console.WriteLine(stackExemplo.Count); // 3
Console.WriteLine(stackExemplo.Peek()); // Output: 3, Pilha = 1,2,3
Console.WriteLine(stackExemplo.Pop()); // Output 3, Pilha = 1,2
Console.WriteLine(stackExemplo.Pop()); // Output 2, Pilha = 1
Console.WriteLine(stackExemplo.Pop()); // Output 1, Pilha = <empty>
Console.WriteLine(stackExemplo.Pop()); // throws exception
```

3.7 Queue<T>

A *Queue<T>* é uma estrutura de FIFO (primeiro que entra é o primeiro a sair). Seus principais métodos são:

- *Enqueue* – Adiciona um elemento no final da fila
- *Dequeue* – Remove o primeiro elemento do início da fila
- *Peek* – Recupera o primeiro elemento do início da fila sem removê-lo
- *Count* – Retorna a quantidade de elementos na fila

A seguir, veja o exemplo de uso dos principais métodos de uma *Queue*:



```
var queueExemplo = new Queue<int>();
queueExemplo.Enqueue(10); // adiciona um elemento
queueExemplo.Enqueue(20); // adiciona um elemento

// Exporta para um array os elementos da fila
int[] data = queueExemplo.ToArray();

Console.WriteLine(queueExemplo.Count); // "2"
Console.WriteLine(queueExemplo.Peek()); // "10"
Console.WriteLine(queueExemplo.Dequeue()); // "10"
Console.WriteLine(queueExemplo.Dequeue()); // "20"
Console.WriteLine(queueExemplo.Dequeue()); // throws an exception (fila vazia)
```

TEMA 4 – CONVERSÕES ENTRE COLLECTIONS

As diferentes coleções do C# permitem que você utilize métodos para conversão entre uma e outra, a depender do tipo. Isso pode ser muito útil quando você precisa manter os elementos já criados em uma coleção, mas mudar a estrutura de dados dessa coleção para outro tipo. Certamente você pode imaginar que, como todas as coleções implementam `IEnumerable<T>`, basta enumerar seus elementos e adicionar um novo Type de coleção (como por exemplo, enumerar um `HashSet` e depois adicionar seus valores em uma `Lista`). Apesar dessa técnica funcionar, ela certamente será muito mais lenta que os métodos que veremos a seguir.

Para entender a base das conversões entre coleções, e porque o processo é eficiente em termos de alocação de memória e performance, precisamos retomar o assunto do início da aula: Arrays. Todas as coleções, invariavelmente, armazenam os dados em memória, utilizando-se de Arrays, o que significa que ele é a base comum para todas as coleções e sequências de dados no C#. Como já vimos, os Arrays apresentam um conceito de implementação e representação nativo na CLR, o que faz com que sejam ideais para isso (Griffiths, 2019, p. 230).

Vamos criar um exemplo em que teremos uma **classe Pessoa**, contendo uma propriedade `Nome` (string) e `Idade` (int). Depois, vamos criar um `Pessoa[]` (array de `Pessoa`), uma `List<Pessoa>`, um `HashSet<Pessoa>` e um `Dictionary<int, Pessoa>`. Nesse exemplo, vamos primeiro popular nosso Array com **5 milhões de registros**. Depois, vamos observar alguns padrões de



conversão entre Arrays e Listas. Na sequência, vamos estudar o método “CopyTo”.

Vejam os a Classe Pessoa que vamos usar em nossos exemplos:

```
public class Pessoa
{
    public int Idade { get; set; }
    public string Nome { get; set; }

    public override string ToString()
    {
        return $"Nome: {Nome} - Idade: {Idade}";
    }
}
```

Observe que fizemos o **override** do método ToString() (método comum a **todos** os objetos no C#). Com a sobrescrita do método ToString, vamos escrever no console o Nome e a Idade dessa “Pessoa”.

Vamos inicializar os Arrays e Listas e criar 5 milhões de objetos Pessoa na memória:

```
var arrayBaseExemplo = new Pessoa[5000000];
var listaExemplo = new List<Pessoa>();
var hashSetExemplo = new HashSet<Pessoa>();
var dicionarioExemplo = new Dictionary<int, Pessoa>();

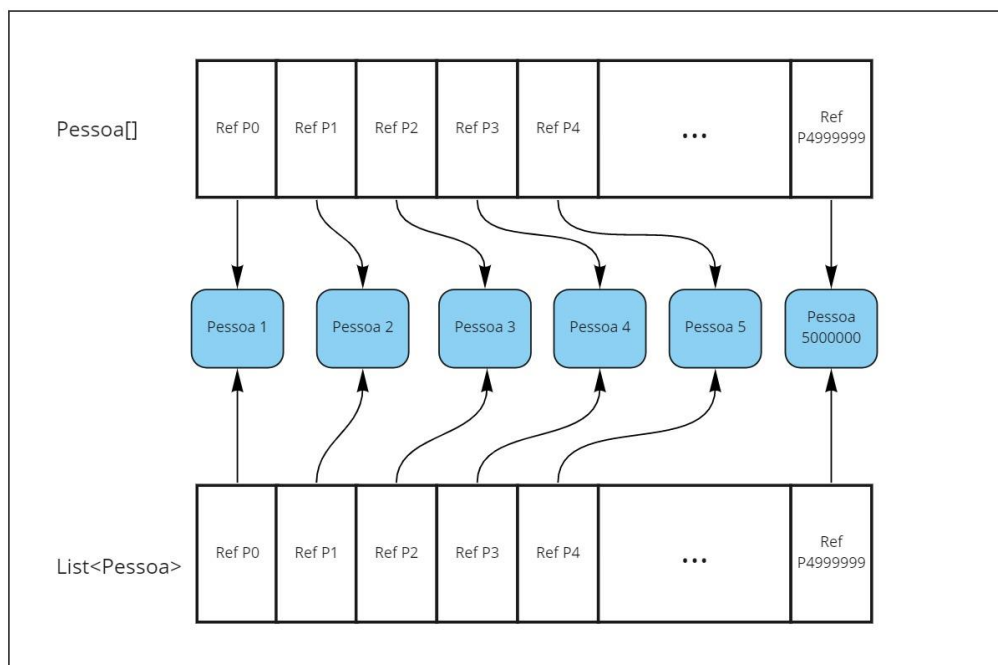
//vamos popular o array base de forma simples:
//interando seus endereços e populando com um número de 1 a 5M
for (int i = 0; i < arrayBaseExemplo.Length; i++)
{
    arrayBaseExemplo[i] = new Pessoa()
    {
        Idade = i,
        Nome = "Pessoa " + i
    };
}
```

Agora que temos um Array com 5 milhões de objetos, vamos fazer a nossa primeira “conversão”. Suponha que você tem um Array de objetos (podem ser int, long, DateTime ou Object – qualquer objeto). Você deseja copiar o conteúdo deste Array para uma List<T>. Uma boa solução é aproveitar o método “AddRange” da List<T>. Esse método copia o conteúdo do Array para a Lista. Uma observação importante: o conteúdo do Array é apenas uma referência ao objeto real.

```
//adiciona todos os elementos do nosso Array para uma List
listaExemplo.AddRange(arrayBaseExemplo);
```

Na imagem a seguir, podemos observar como é tratada a referência de objetos nas coleções do C#. Observe que o objeto Pessoa (em azul) é referenciado pelo Array. Logo, seu espaço de memória não está dentro do Array. O Array e a Lista apenas contêm referências para o objeto, que por sua vez pode conter referências para propriedades internas, caso elas sejam “*Reference Types*”:

Figura 2 – Objetos em coleções do C#



Para exemplificar esse ponto, vejamos o código a seguir, em que atualizaremos o valor do elemento de index 1000 para o nome “atualizado”. Após

```
//como a lista armazena as referências para tipos de Reference Types
//se modificarmos o valor de um elemento de nosso array
//o elemento que a lista referencia também será alterado

//Observe que o elemento 1000 escreverá no console: Nome: Pessoa 1000 - Idade: 1000
Console.WriteLine(arrayBaseExemplo[1000]); // Nome: Pessoa 1000 - Idade: 1000

//Vamos atualizar o nome da pessoa
arrayBaseExemplo[1000].Nome = "atualizado";

//Observe que a lista e o array apontam para a mesma referência de objetos
Console.WriteLine(arrayBaseExemplo[1000]); //Nome: atualizado - Idade: 1000
Console.WriteLine(listaExemplo[1000]); // Nome: atualizado - Idade: 1000
```



essa operação, se o valor desse objeto for impresso no console, veremos que ambas as coleções apontam para o mesmo objeto:

Agora, vamos copiar o conteúdo do nosso Array para um HashSet:

```
//Inicializa o HashSet usando uma lista (ou qualquer IEnumerable)
hashSetExemplo = new HashSet<Pessoa>(listaExemplo);
```

Observe que HashSets não permitem adicionar valores em massa, como a Lista. Além disso, a estrutura de um HashSet depende de um conceito matemático chamado Hashtable. Porém, o HashSet nos provê um modo de inicializar um HashSet<T> utilizando uma ICollection<T>. Internamente, ele fará o mesmo que a lista, copiando a referência de array e gerando um Hashtable interno. No exemplo a seguir, vemos que o HashSet mantém a referência para o objeto, mesmo com uma estrutura de Array internamente diferente:

```
//vamos alterar o nome e idade da pessoa 1000
var pessoa1000 = arrayBaseExemplo[1000];
pessoa1000.Idade = -1;
pessoa1000.Nome = "atualizado 1000";

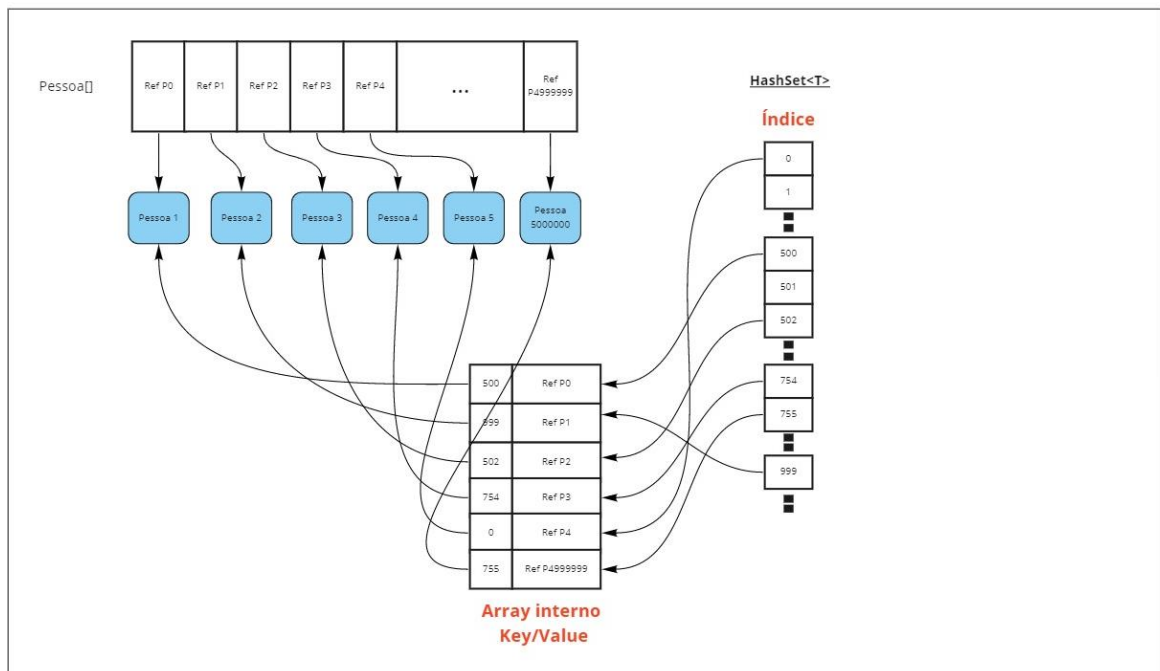
//Veja que a referência utilizada pela lista já foi alterada
Console.WriteLine(listaExemplo[1000]); // Nome: atualizado 1000 - Idade: -1

//Como o HashSet não fornece uma propriedade indexadora (acessar via [])
//por questões de conceito, vamos usar o TryGetValue para
//obter o valor do elemento.
if (hashSetExemplo.TryGetValue(pessoa1000, out Pessoa pessoaOutput))
{
    //Observe que o HashSet também manteve a referência para o mesmo objeto
    //mesmo tendo uma estrutura de armazenamento de dados diferente de listas

    Console.WriteLine(pessoaOutput); // Nome: atualizado 1000 - Idade: -1
}
```

Na imagem, podemos ver como a estrutura HashSet mantém os seus dados na memória. Ela utiliza basicamente dois arrays internos: um para armazenar um índice do objeto e outro para armazenar a referência. O segundo Array também guarda o valor do índice dentro de uma struct similar a do Dictionary (KeyValuePair). Apesar de parecer estranho, esse conceito está fundamentado no princípio de gerar elevada performance na busca, através de uma função de dispersão e de uma estrutura de índices.

Figura 3 – Estrutura HashSet



Observe que a estrutura de índices nem sempre aponta para um valor. Isso ocorre porque a função de dispersão, que é executada sempre que se adiciona um elemento em um HashSet, gera um índice para esse elemento com base em um procedimento matemático, e não em uma sequência, como no caso de Listas e Arrays. Esse é o motivo pelo qual não se pode buscar um elemento no HashSet com base em seu índice. Afinal, esse índice é controlado internamente pela coleção, e ela não ocupa todas as suas posições de forma linear, e sim de forma dispersa (Livro C# 8.0).

Inicializar um Dictionary<TKey, TValue> é um pouco mais complicado. Enquanto o HashSet gera índices internos para seus objetos, o Dictionary obriga você a fornecer a chave do “índice”, passando um **struct KeyValuePair** como parâmetro para cada inserção na coleção. Vamos fazer isso enumerando a nossa lista e inserindo-a no dicionário. Na sequência, veremos como fazer isso usando funções de extensão que **facilitam muito** a nossa vida no dia a dia do C#.



```
//Inicializando um Dictionary
dicionarioExemplo = new Dictionary<int, Pessoa>(listaExemplo.Count);

foreach (var pessoa in listaExemplo)
{
    //estamos usando a "idade da pessoa" como TKey (int)
    //para adicionar no dicionário a KeyValuePair<int,Pessoa> (idade, pessoa)
    //onde o valor da idade esta representando o índice/chave do dicionário
    //lembre-se: caso hajam valores duplicados o dicionário irá lançar uma exceção

    dicionarioExemplo.Add(pessoa.Idade, pessoa);
}

//Acessando a pessoa1000
Console.WriteLine(dicionarioExemplo[pessoaOutput.Idade]); // Nome: atualizado 1000
- Idade: -1
```

Sem o uso de funções de extensão, não temos muitas alternativas a não ser a enumeração do Array e a respectiva carga na coleção Dictionary, como mostra o trecho do código supracitado.

TEMA 5 – MÉTODOS DE EXTENSÃO X COLEÇÕES

No C#, há um recurso muito poderoso e útil, introduzido desde a sua versão 3.5, chamado *Extensions Methods*. Além do LINQ, tema que veremos mais adiante, essa versão trouxe muitas novidades para trabalhar com coleções no C#. Aproveitando-se de conceitos de outras linguagens funcionais e dinâmicas (como Python e Haskell), os métodos de extensão garantem ao C# um grande poder de extensão de comportamentos.

Os métodos de extensão permitem que uma classe existente seja estendida com novos métodos, sem alterar a sua definição original. Um método de extensão é um método estático de uma classe estática, em que o modificador “**this**” é aplicado ao primeiro parâmetro. O tipo do primeiro parâmetro será o Type a ser estendido (Albahari; Albahari, 2017, p. 180; Griffiths, 2019, p. 179). Por exemplo:



```
public static class StringHelper
{
    public static bool IsCapitalized(this string s)
    {
        if (string.IsNullOrEmpty(s))
        {
            return false;
        }

        return char.IsUpper(s[0]);
    }
}
```

O trecho desse código tem, à primeira vista, algumas notações estranhas, como: uso do modificador “**static**” na classe e do modificador “**this**” no método *IsCapitalized*. Essas são as obrigações de que você precisa para declarar um método de extensão.

Esse método irá, a partir de agora, ser listado como método “normal” dentro de uma string. Seu utilizador poderá usufruir dele como se a própria classe string apresentasse o método:

```
bool isCaps = "aula 2".IsCapitalized();

Console.WriteLine(isCaps); // false

string exemplo2 = "String Caps";

if (exemplo2.IsCapitalized())
{
    Console.WriteLine("exemplo 2 está em Maiúscula!");
}
```

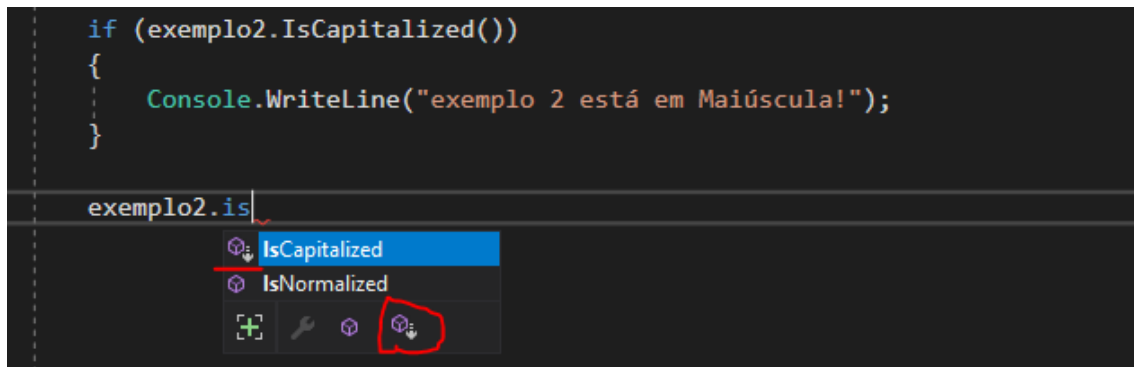
Perceba que utilizamos o método como se a própria classe string já o possuísse. Essa é a grande vantagem de utilizar métodos de extensão. Você pode criar uma classe estática em outro arquivo, até mesmo em outro projeto/dll, e utilizar os métodos de extensão definidos dentro de uma classe já existente, sua ou definida pela linguagem, como foi o caso da string no exemplo.

Você pode também encadear o uso de métodos de extensão ou usá-los em sua forma estática comum. Veja, no trecho a seguir, que x e y são equivalentes:



```
string x = "exemplo".Pluralize().Capitalize();
string y = StringHelper.Capitalize(StringHelper.Pluralize("exemplo"));
```

Você pode reconhecer um método de extensão com facilidade pelo seguinte ícone na IDE (Visual Studio / VS Code):



5.1 Métodos de extensão de collections

Com a chegada da versão 3.5 do framework .NET, por volta de 2008, muitos métodos de extensão foram criados para facilitar o uso das collections. Vários desses métodos na conversão entre os principais tipos de coleção, como no exemplo a seguir:

```
//Lista de strings
var listaExemplo = new List<string>();

//Vamos carregar a lista com 10 objetos
for (int i = 0; i < 10; i++)
{
    listaExemplo.Add("item: " + i);
}

//Conversão de uma lista para um HashSet utilizando métodos de extensão
var hashSetExemplo = listaExemplo.ToHashSet();
```

Observe o método *ToHashSet()*. Ele é um método de extensão. Eles facilitam muito a nossa vida para operações rotineiras. Quando se trata de coleções, os métodos de extensão do próprio .NET ajudam quando você precisa migrar de um tipo para outro tipo de coleção. Quando o tipo for muito específico, e não puder prover um método *ToList* por algum motivo, certamente você poderá



convertê-lo para um Array e depois para uma List. Veja a seguir como é simples converter uma Stack (pilha) em uma List (Griffiths, 2019, p. 180):

```
var stack = new Stack<string>(listaExemplo);  
stack.ToList();
```

Neste caso, podemos observar dois grandes recursos da linguagem operando juntos: *Generics* e *Extension Methods*. Isso porque quase todos os métodos de extensão, comuns às coleções, estendem o comportamento de `IEnumerable<T>`. Como praticamente todas as coleções implementam a interface `IEnumerable<T>`, você pode convertê-las facilmente de uma para outra.

Os métodos de extensão precisam que o namespace em que estão definidos seja referenciado no “**using**” da classe que pretende utilizá-los. Dessa forma, a IDE não irá sugerir todos os métodos de extensão existentes logo de primeira. Caberá a você fazer **uso dos namespaces**, considerando onde eles se encontram, além de adicionar novos conforme a necessidade de novos métodos existentes (Griffiths, 2019, p. 180).

Para facilitar as extensões entre coleções, grande parte dos métodos de extensão de coleções está definida no namespace **System.Linq**. Basta você referenciar esse namespace para se utilizar de vários métodos, desde conversão entre coleções, até novos métodos de sumarização, consulta, contagem e muito mais! Vamos explorar todos esses recursos nos próximos conteúdos!

FINALIZANDO

Nesta aula, conhecemos mais sobre o C#, em especial o uso de *Generics* e de coleções de linguagem. Ambos os recursos são comumente usados no dia a dia do desenvolvimento.

É importante lembrar que a implementação do conceito de *Generics* não é exclusiva do C#, encontrando equivalências no Java e Kotlin, por exemplo. É um recurso natural de linguagens orientadas a objeto “tipadas”. Portanto, é fundamental dominar o conceito. Você poderá exercitá-lo em nossas aulas práticas e também no seu dia a dia, uma vez que é impraticável trabalhar sem ele em aplicações reais.

Da mesma forma, o uso de coleções é uma tarefa rotineira e diária na vida de um desenvolvedor .NET. Conhecer e sentir-se confortável com os diferentes



tipos de coleções existentes, inclusive conversão entre elas, enumeração de seus valores etc., é fundamental para todo desenvolvedor.

Nesta aula, você teve uma base sólida desses conceitos. Recomendamos ainda que você consulte as referências do quadro a seguir e da listagem final, e que explore esses conceitos ainda mais no seu dia a dia, pois será importante dominar a base da linguagem para migrar de plataforma (Web, Desktop e Mobile) sem grandes dificuldades.

Saiba mais

MICROSOFT, **Generics in .NET**. Disponível em: <<https://docs.microsoft.com/pt-br/dotnet/standard/generics>>. Acesso em 05 de maio de 2021.

MICROSOFT, **Coleções (C#)**. Disponível em: <<https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/collections>>. Acesso em 15 de maio de 2021.

MICROSOFT, **Coleções e estrutura de dados**. Disponível em: <<https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/collections>>. Acesso em 15 de maio de 2021.

MICROSOFT, **Quando usar coleções genéricas**. Disponível em: <<https://docs.microsoft.com/pt-br/dotnet/standard/collections/when-to-use-generic-collections>>. Acesso em 15 de maio de 2021.

MICROSOFT, **Hashtable**. Disponível em: <<https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable>>. Acesso em 15 de maio de 2021.

MICROSOFT, **Learn C# (Official Microsoft Guide)**. Disponível em: <<https://docs.microsoft.com/pt-br/users/dotnet/collections/yz26f8y64n7k07>>. Acesso em 15 de maio de 2021.



REFERÊNCIAS

ALBAHARI, J.; ALBAHARI, B. **C# 7.0 in a Nutshell**. 7. ed. United States of America: O'Reilly Media Inc, 2017.

GRIFFITHS, I. **Programming C# 8.0**. 2.ed. United States of America: O'Reilly Media Inc, 2019.