# COLLECTIONS

CAPÍTULO 17

Quando uma aplicação precisa manipular uma quantidade grande de dados, ela deve utilizar alguma estrutura de dados. Podemos dizer que a estrutura de dados mais básica do C# são os arrays.

Muitas vezes, trabalhar diretamente com arrays não é simples dado as diversas limitações que eles possuem. A limitação principal é a capacidade fixa, um array não pode ser redimensionado. Se todas as posições de um array estiverem ocupadas não podemos adicionar mais elementos. Normalmente, criamos um outro array com maior capacidade e transferimos os elementos do array antigo para o novo.

Além disso, adicionar ou remover elementos provavelmente gera a necessidade de deslocar parte do conteúdo do array.

As dificuldades do trabalho com array podem ser superadas com estruturas de dados mais sofisticadas. Na biblioteca do C#, há diversas estruturas de dados que facilitam o trabalho do desenvolvedor.

## Listas

As listas são estruturas de dados de armazenamento sequencial assim como os arrays. Mas, diferentemente dos arrays, as listas não possuem capacidade fixa o que facilita bastante o trabalho.

`IList` é a interface C# que define os métodos que uma lista deve implementar. A principal implementação dessa interface é a classe `ArrayList`.

```
1 ArrayList arrayList = new ArrayList();
```

*Código C# 17.1: Criando uma lista*

Podemos aplicar o polimorfismo e referenciar objetos criados a partir da classe: `ArrayList` como `IList`.

```
1 IList list = new ArrayList();
```

*Código C# 17.2: Aplicando polimorfismo*

### Método: Add(object)

O método `Add(object)` adiciona uma referência no final da lista e aceita referências de qualquer tipo.

```
1 IList list = ...
2
3 list.Add(258);
```

```
4  list.Add("Rafael Cosentino");
5  list.Add(1575.76);
6  list.Add("Marcelo Martins");
```

*Código C# 17.3: Adicionando elementos no final de uma lista*

### Método: Insert(int, object)

O método `Insert(int, object)` adiciona uma referência em uma determinada posição da lista. A posição passada deve ser positiva e menor ou igual ao tamanho da lista.

```
1  IList list = ...
2
3  list.Insert(0, "Jonas Hirata");
4  list.Insert(1, "Rafael Cosentino");
5  list.Insert(1, "Marcelo Martins");
6  list.Insert(2, "Thiago Thies");
7  //tamanho da lista 4
8  //ordem:
9  // 1. Jonas Hirata
10 // 2. Marcelo Martins
11 // 3. Thiago Thies
12 // 4. Rafael Cosentino
```

*Código C# 17.4: Adicionando elementos em qualquer posição de uma lista*

### Propriedade: Count

A propriedade `Count` informa a quantidade de elementos armazenado na lista.

```
1  IList list = ...
2
3  list.Add("Jonas Hirata");
4  list.Add("Rafael Cosentino");
5  list.Add("Marcelo Martins");
6  list.Add("Thiago Thies");
7
8  // quantidade = 4
9  int quantidade = list.Count;
```

*Código C# 17.5: Recuperando a quantidade de elementos de uma lista.*

### Método: Clear()

O método `Clear()` remove todos os elementos da lista.

```
1  IList list = ...
2
3  list.Add("Jonas Hirata");
4  list.Add("Rafael Cosentino");
5  list.Add("Marcelo Martins");
6  list.Add("Thiago Thies");
7
8  // quantidade = 4
9  int quantidade = list.Count;
10
11 list.Clear();
12
13 // quantidade = 0
14 quantidade = list.Count;
```

*Código C# 17.6: Eliminando todos os elementos de uma lista*

### Método: Contains(object)

Para verificar se um elemento está contido em uma lista, podemos utilizar o método `Contains(object)`

```
IList list = ...

list.Add("Jonas Hirata");
list.Add("Rafael Cosentino");

// x = true
bool x = list.Contains("Jonas Hirata");

// x = false
x = list.Contains("Daniel Machado");
```

*Código C# 17.7: Verificando se um elemento está em uma lista*

### Método: Remove(object)

Podemos retirar elementos de uma lista através do método `Remove(object)`. Este método remove a primeira ocorrência do elemento passado como parâmetro.

```
IList list = ...

list.Add("Jonas Hirata");

// x = true
bool x = list.Contains("Jonas Hirata");

list.Remove("Jonas Hirata");

// x = false
x = list.Contains("Jonas Hirata");
```

*Código C# 17.8: Removendo um elemento de uma lista*

### Método: RemoveAt(int)

Outra maneira para retirar elementos de uma lista através do método `RemoveAt(int)`.

```
IList list = ...

list.Add("Jonas Hirata");

// x = true
bool x = list.Contains("Jonas Hirata");

list.RemoveAt(0);

// x = false
x = list.Contains("Jonas Hirata");
```

*Código C# 17.9: Removento um elemento de uma lista por posição*

### Propriedade: Item

Para recuperar um elemento de uma determinada posição de uma lista podemos utilizar a propriedade `Item`. Com esta propriedade, podemos utilizar a seguinte sintaxe para acessar um elemento numa determinada posição: `myList[posicao]`

```
IList list = ...
```

```
2
3  list.Add("Jonas Hirata");
4
5  // nome = "Jonas Hirata"
6  string nome = list[0];
```

*Código C# 17.10: Acessando o elemento de uma determinada posição de uma lista*

### Método: IndexOf(object)

Para descobrir o índice da primeira ocorrência de um determinado elemento podemos utilizar o método `IndexOf(object)`.

```
1  IList list = ...
2
3  list.Add("Jonas Hirata");
4
5  // indice = 0
6  int indice = list.IndexOf("Jonas Hirata");
```

*Código C# 17.11: Descobrindo o índice da primeira ocorrência de um elemento em uma lista*

## Exercícios de Fixação

**1** Crie um projeto do tipo Console Application no *Microsoft Visual C# Express* chamado **Collections**.

**2** Vamos calcular o tempo das operações de uma ArrayList.

```
1  using System;
2  using System.Collections;
3  using System.Diagnostics;
4
5  public class TestaAdicionaNoFinal
6  {
7    static void Main()
8    {
9      ArrayList arrayList = new ArrayList();
10
11     long tempo = TestaAdicionaNoFinal.AdicionaNoFinal(arrayList);
12     Console.WriteLine("ArrayList: " + tempo + "ms");
13
14   }
15
16   public static long AdicionaNoFinal(IList lista)
17   {
18     Stopwatch sw = new Stopwatch();
19
20     sw.Start();
21     int size = 100000;
22
23     for (int i = 0; i < size; i++)
24     {
25       lista.Add(i);
26     }
27
28     sw.Stop();
29
30     return sw.ElapsedMilliseconds;
```

```
31    }
32  }
```

*Código C# 17.12: TestaAdicionaNoFinal.cs*

```
1  using System;
2  using System.Collections;
3  using System.Diagnostics;
4
5  public class TestaAdicionaNoComeco
6  {
7    static void Main()
8    {
9      ArrayList arrayList = new ArrayList();
10
11     long tempo = TestaAdicionaNoComeco.AdicionaNoComeco(arrayList);
12     Console.WriteLine("ArrayList: " + tempo + "ms");
13   }
14
15   public static long AdicionaNoComeco(IList lista)
16   {
17     Stopwatch sw = new Stopwatch();
18     sw.Start();
19     int size = 100000;
20
21     for (int i = 0; i < size; i++)
22     {
23       lista.Insert(0, i);
24     }
25
26     sw.Stop();
27
28     return sw.ElapsedMilliseconds;
29   }
30 }
```

*Código C# 17.13: TestaAdicionaNoComeco.cs*

3 Teste o desempenho para remover elementos do começo ou do fim da ArrayList.

## Generics

As listas armazenam referências de qualquer tipo. Dessa forma, quando recuperamos um elemento de uma lista temos que trabalhar com referências do tipo `object`.

```
1  IList list = ...
2
3  list.Add("Rafael Cosentino");
4
5  foreach(object x in list)
6  {
7    Console.WriteLine(x);
8  }
```

*Código C# 17.14: Percorrendo uma lista que armazena qualquer tipo de referência*

Porém, normalmente, precisamos tratar os objetos de forma específica pois queremos ter acesso aos métodos específicos desses objetos. Nesses casos, devemos fazer casting nas referências.

```
1  IList list = ...
```

```
2
3  list.Add("Rafael Cosentino");
4
5  foreach(object x in list)
6  {
7    string s = (string)x;
8    Console.WriteLine(s.ToUpper());
9  }
```

*Código C# 17.15: Percorrendo uma lista que armazena strings*

O casting de referência é arriscado pois em tempo de compilação não temos garantia que ele está correto. Dessa forma, corremos o risco de obter um erro de execução.

Para ter certeza da tipagem dos objetos em tempo de compilação, devemos aplicar o recurso do **Generics**. Com este recurso podemos determinar o tipo de objeto que queremos armazenar em uma coleção no momento em que ela é criada. A partir daí, o compilador não permitirá que elementos não compatíveis com o tipo escolhido sejam adicionados na coleção. Isso garante o tipo do elemento no momento em que ele é recuperado da coleção e elimina a necessidade de casting.

As classes que contém o recurso **Generics** fazem parte do namespace *System.Collections.Generic*. A classe genérica equivalente a `ArrayList` é a `List`. Outra implementação importante de listas genéricas é a `LinkedList`.

```
1  List<string> arrayList = new List<string>();
2  LinkedList<string> linkedList = new LinkedList<string>();
```

*Código C# 17.16: Criando listas parametrizadas*

```
1  List<string> arrayList = new List<string>();
2
3  arrayList.Add("Rafael Cosentino");
4
5  foreach(string x in arrayList)
6  {
7    Console.WriteLine(x.ToUpper());
8  }
9
10 LinkedList<string> linkedList = new LinkedList<string>();
11
12 linkedList.AddLast("Rafael Cosentino");
13
14 foreach(string x in linkedList)
15 {
16   Console.WriteLine(x.ToUpper());
17 }
```

*Código C# 17.17: Trabalhando com listas parametrizadas*

## Benchmarking

As duas principais implementações de listas genéricas em C# possuem desempenho diferentes para cada operação. O desenvolvedor deve escolher a implementação de acordo com a sua necessidade.

| Operação | List | LinkedList |
|---|---|---|
| Adicionar ou Remover do final da lista | ☺ | ☺ |
| Adicionar ou Remover do começo da lista | ☹ | ☺ |
| Acessar elementos pela posição | ☺ | ☹ |

## Exercícios de Fixação

4. Crie um projeto do tipo Console Application no *Microsoft Visual C# Express* chamado **Generic**.

5. Vamos calcular o tempo das operações das classes *List* e *LinkedList*.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

public class TestaAdicionaNoFinal
{
  static void Main()
  {
    List<int> arrayList = new List<int>();

    long tempo = TestaAdicionaNoFinal.AdicionaNoFinal(arrayList);
    Console.WriteLine("ArrayList: " + tempo + "ms");

    LinkedList<int> linkedList = new LinkedList<int>();

    tempo = TestaAdicionaNoFinal.AdicionaNoFinal(linkedList);
    Console.WriteLine("LinkedList: " + tempo + "ms");
  }

  public static long AdicionaNoFinal(ICollection<int> lista)
  {
    Stopwatch sw = new Stopwatch();

    sw.Start();
    int size = 100000;

    for (int i = 0; i < size; i++)
    {
      lista.Add(i);
    }

    sw.Stop();

    return sw.ElapsedMilliseconds;
  }
}
```

*Código C# 17.18: TestaAdicionaNoFinal.cs*

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

public class TestaAdicionaNoComeco
{
  static void Main()
```

```
8   {
9     List<int> arrayList = new List<int>();
10
11    long tempo = TestaAdicionaNoComeco.AdicionaNoComecoArrayList(arrayList);
12    Console.WriteLine("ArrayList: " + tempo + "ms");
13
14    LinkedList<int> linkedList = new LinkedList<int>();
15
16    tempo = TestaAdicionaNoComeco.AdicionaNoComecoLinkedList(linkedList);
17    Console.WriteLine("LinkedList: " + tempo + "ms");
18
19  }
20
21  public static long AdicionaNoComecoArrayList(List<int> lista)
22  {
23    Stopwatch sw = new Stopwatch();
24    sw.Start();
25    int size = 100000;
26
27    for (int i = 0; i < size; i++)
28    {
29      lista.Insert(0, i);
30    }
31
32    sw.Stop();
33
34    return sw.ElapsedMilliseconds;
35  }
36
37  public static long AdicionaNoComecoLinkedList(LinkedList<int> lista)
38  {
39    Stopwatch sw = new Stopwatch();
40    sw.Start();
41    int size = 100000;
42
43    for (int i = 0; i < size; i++)
44    {
45      lista.AddFirst(i);
46    }
47
48    sw.Stop();
49
50    return sw.ElapsedMilliseconds;
51  }
52 }
```

*Código C# 17.19: TestaAdicionaNoComeco.cs*

6 Teste o desempenho para remover elementos do começo ou do fim das principais listas.

## Conjuntos

Os conjuntos diferem das listas pois não permitem elementos repetidos e não possuem ordem. Como os conjuntos não possuem ordem as operações baseadas em índice que existem nas listas não aparecem nos conjuntos.

`ISet` é a interface genérica C# que define os métodos que um conjunto deve implementar. A principal implementação da interface `ISet` é: `HashSet`.

## Coleções

Há semelhanças conceituais entre os conjuntos e as listas por isso existe uma super interface genérica chamada `ICollection` para as interfaces genéricas `IList` e `ISet`.

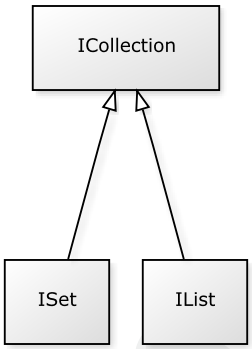


*Figura 17.1: Coleções*

Dessa forma, podemos referenciar como `ICollection` qualquer lista ou conjunto.

```
1 ICollection<string> conjunto = new HashSet<string>();
2 ICollection<string> lista = new List<string>();
```

*Código C# 17.20: Aplicando polimorfismo*

## Laço foreach

As listas podem ser iteradas com um laço `for` tradicional.

```
1 IList<string> lista = new List<string>();
2
3 for(int i = 0; i < lista.Count; i++)
4 {
5   string x = lista[i];
6 }
```

*Código C# 17.21:* `for` *tradicional*

Porém, como os conjuntos não são baseados em índice eles não podem ser iterados com um laço `for` tradicional. A maneira mais eficiente para percorrer uma coleção é utilizar um laço **foreach**.

```
1 ICollection<string> colecao = ...
2
3 foreach(string x in colecao)
4 {
5
6 }
```

*Código C# 17.22:* `foreach`

O `foreach` é utilizado para percorrer os elementos da coleção e recuperar a informação que você deseja, mas não é possível utilizá-lo para adicionar ou remover elementos, para estes casos você deve utilizar o `for`.

## Exercícios de Fixação

7 Vamos comparar o tempo do método `Contains()` das listas e dos conjuntos.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

public class TestaContains
{
  static void Main()
  {
    List<int> arrayList = new List<int>();
    HashSet<int> hashSet = new HashSet<int>();

    long tempo = TestaContains.Contains(arrayList);
    Console.WriteLine("ArrayList: " + tempo + "ms");

    tempo = TestaContains.Contains(hashSet);
    Console.WriteLine("HashSet: " + tempo + "ms");

  }

  public static long Contains(ICollection<int> colecao)
  {
    int size = 100000;

    for (int i = 0; i < size; i++)
    {
      colecao.Add(i);
    }

    Stopwatch sw = new Stopwatch();
    sw.Start();

    for (int i = 0; i < size; i++)
    {
      colecao.Contains(i);
    }

    sw.Stop();

    return sw.ElapsedMilliseconds;
  }
}
```

*Código C# 17.23: TestaContains.cs*