

# Coleções

a

## Entendendo a necessidade de coleções

Utilizamos as coleções (ou estruturas de dados) quando temos muitos dados e queremos trabalhar com eles de forma agrupada. Assim, evitamos casos de precisar criar e gerenciar muitas variáveis de uma vez, como no exemplo abaixo:

```
string funcionario = “João”;
```

```
string funcionario1 = “Maria”;
```

```
string funcionario2 = “Cláudia”;
```

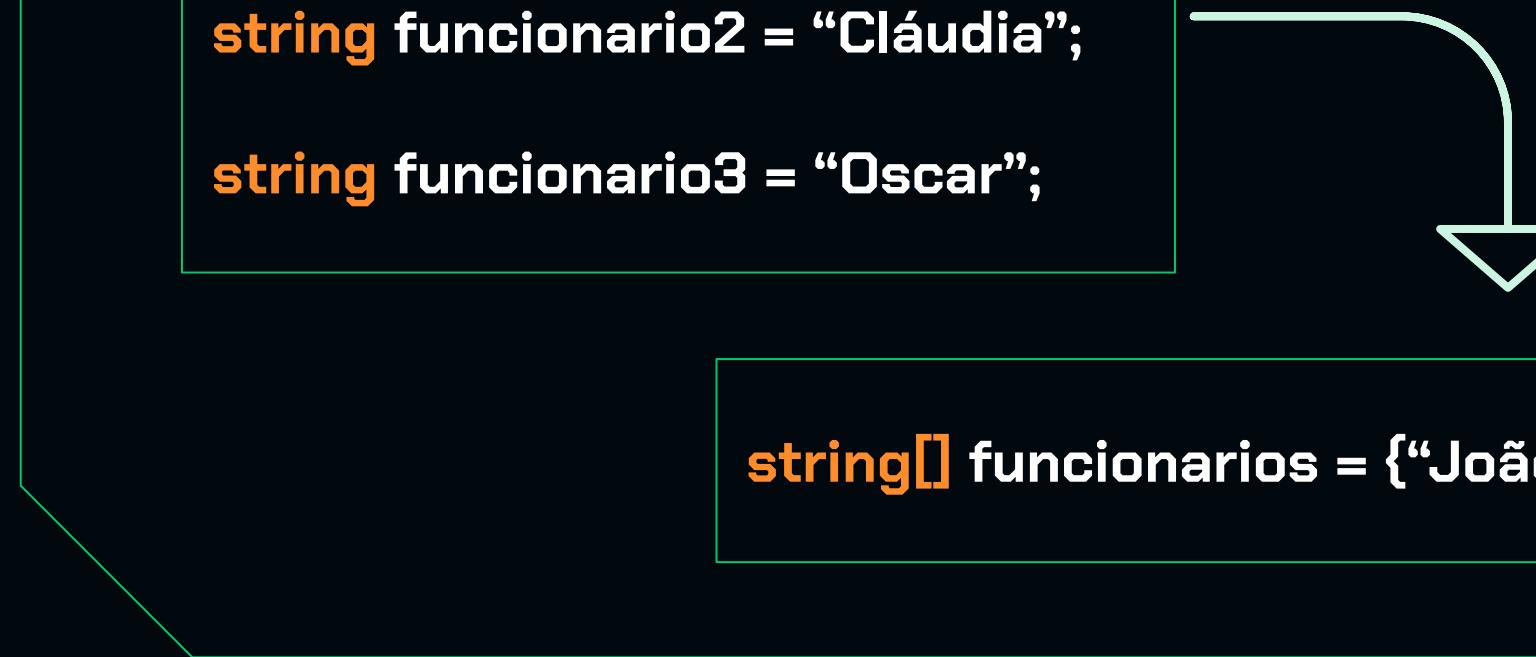
```
string funcionario3 = “Oscar”;
```

...

# Arrays

A coleção mais simples que podemos utilizar para armazenar vários dados é o array:

```
string funcionario = "João";  
  
string funcionario1 = "Maria";  
  
string funcionario2 = "Cláudia";  
  
string funcionario3 = "Oscar";
```



Os dois códigos armazenam exatamente os mesmos dados. A diferença é que nos arrays não precisamos criar "trocentas" variáveis diferentes, criamos apenas **uma variável** que tornará possível guardar **todos os dados de uma vez**.

```
string[] funcionarios = {"João", "Maria", "Cláudia", "Oscar"};
```

# Arrays

Para indicar ao compilador que vamos trabalhar com variáveis que são arrays, utilizamos os colchetes (`[]`) logo depois do tipo da variável. Podemos inicializar um array como indicado abaixo:

Uso dos colchetes  
logo depois do tipo  
da variável

.....→ **string[] funcionarios = {"João", "Maria", "Cláudia", "Oscar"};**

Nesse estilo, já declaramos nosso array e dizemos como ele será preenchido de uma vez. Aqui, o compilador já infere que o tamanho do array é 4.

# Arrays

Também podemos inicializar um array assim:

Uso dos colchetes  
logo depois do tipo  
da variável

Podemos usar colchetes  
depois do `new` para indicar  
o tamanho do array

```
string[] funcionarios = new string[4];  
  
funcionarios[0] = "João";  
  
funcionarios[1] = "Maria";  
  
....
```

Já nessa forma, apenas dizemos para o compilador **reservar 4 posições do tipo string na memória**. A princípio, elas não armazenam nada. Mas depois da inicialização, podemos escolher a posição queremos preencher, usando índices **a partir de 0**. No exemplo, temos 4 posições no array, e os índices das posições vão de 0 até 3. Para preencher o array `funcionarios` na posição 0, utilizamos `funcionarios[0]`.

# Arrays

Além de acessar as posições com colchetes para inicializar um valor, também localizar um valor específico com os colchetes

```
string[] funcionarios = {"João",  
"Maria", "Cláudia", "Oscar"};  
  
Console.WriteLine(funcionarios[1]);
```

No código ao lado, estamos imprimindo o valor específico do array na posição 1. Dessa forma, o resultado que irá aparecer na tela será "Maria".

Ao realizar essa operação, dizemos que estamos **indexando um array**. Isso porque estamos fazendo uma **busca dado o índice**.

# Arrays

```
string[] funcionarios = { "João", "Maria", "Cláudia", "Oscar" };

for (int i = 0; i < funcionarios.Length; i++)
{
    Console.WriteLine(funcionarios[i]);
}
```

Podemos utilizar a busca por índices para percorrer um array, num **for** tradicional.

No código ao lado, estamos imprimindo o valor específico do array na posição 1. Dessa forma, o resultado que irá aparecer

```
string[] funcionarios = { "João", "Maria", "Cláudia", "Oscar" };

foreach (string nome in funcionarios)
{
    Console.WriteLine(nome);
}
```

Mas caso queiramos usar o foreach, também é possível.

# Arrays

Também é possível tentar localizar qual o índice de determinado elemento.

```
string[] funcionarios = {"João", "Maria", "Cláudia", "Oscar";  
int indice = Array.IndexOf(funcionarios, "Oscar");  
Console.WriteLine($"Posição do Oscar = {indice}");
```

No código ao lado, estamos utilizando a classe específica **Array**, onde há diversos métodos úteis para trabalhar com muitos dados. Um deles é o **IndexOf**.

Outros métodos úteis dessa classe são:  
**Sort()**, para ordenar os elementos que estão no array e **Reverse()**, para inverter os elementos.

## Limitações de Arrays

Arrays são coleções com estrutura limitada: uma vez que declaramos o tamanho de um array, ele terá exatamente esse tamanho para sempre. Por isso, adicionar e remover elementos são operações bastante trabalhosas.

```
string[] funcionarios = { "João", "Maria", "Cláudia", "Oscar" };

string[] novoArray = new string[5];
Array.Copy(funcionarios, novoArray, funcionarios.Length);
foreach (string nome in novoArray)
{
    Console.WriteLine(nome);
}
novoArray[4] = "Joana";
```

Ao lado, temos uma simulação de inserção. Repare que temos de criar um novo array para conseguir guardar um novo elemento. A remoção seria uma tarefa igualmente complicada.

## Coleções genéricas

Dadas as limitações dos arrays, o C# tem um namespace chamado **System.Collections.Generic**, onde podemos trabalhar com **coleções genéricas**.

A grande vantagem das coleções genéricas é a flexibilidade no tamanho. Assim, conseguimos inserir e remover os elementos de forma fácil e rápida. Além disso, elas também têm métodos mais sofisticados para trabalhar com os dados.

Iremos explorar as 3 coleções genéricas mais usadas em C#: **List**, **HashSet** e **Dictionary**. Cada uma delas tem suas particularidades, que fazem com que sejam mais (ou menos) adequadas para resolver um problema específico.

## Listas

As listas são representadas pela classe `List` e têm um comportamento muito parecido com o dos arrays.

A diferença é que, dada a implementação, os **arrays** se saem melhor quando trabalhamos com **busca por índices**, enquanto as listas são melhores para trabalharmos com tamanhos dinâmicos de dados (podendo **inserir e remover elementos facilmente**).

Por ser uma coleção genérica, ao declarar uma `List`, devemos dizer qual o tipo de dados ela irá armazenar. Esse tipo é passado entre os sinais `<>`.

```
.....  
List<string> listaFuncionarios = new List<string>  
{  
    "João", "Maria", "Cláudia", "Oscar"  
};
```

# Listas

Há duas formas de criar uma lista:

A lista pode ser inicializada já com alguns valores, caso os dados estejam disponíveis. Por mais que seja inicializada com 4 nomes no exemplo, a lista pode ser expandida depois, não tendo tamanho fixo.

```
List<string> listaFuncionarios = new List<string>
{
    "João", "Maria", "Cláudia", "Oscar"
};
```

```
List<string> listaFuncionarios = new List<string>();
listaFuncionarios.Add("João");
listaFuncionarios.Add("Maria");
listaFuncionarios.Add("Cláudia");
listaFuncionarios.Add("Oscar");
```

A lista também pode ser inicializada vazia, usando os (). Para adicionar valores depois, utilizamos o método Add()

# Listas

Listas podem ser percorridas da mesma forma que os arrays, com for e foreach:

```
List<string> listaFuncionarios = new List<string>
{
    "João", "Maria", "Cláudia", "Oscar"
};

for(int i = 0; i < listaFuncionarios.Count; i++)
{
    Console.WriteLine(listaFuncionarios[i]);
}

foreach (string nome in listaFuncionarios)
{
    Console.WriteLine(nome);
}
```

Repare que também conseguimos utilizar os colchetes para indexar uma lista. Embora de forma visual seja igual ao que fazemos em arrays, “por baixo dos panos” essa **indexação é diferente e menos eficiente nas listas**. Assim, se o seu foco é fazer buscas indexadas, sem necessidade de tamanho dinâmico, escolha trabalhar com arrays.

# Listas

Para adicionar elementos na lista, utilizamos o método `Add()`. E para remover elementos, usamos o método `Remove()`.

Também podemos trabalhar com diversos outros métodos das listas, como `Sort()`, `Reverse()`, `IndexOf()`...

```
listaFuncionarios.Add("Joana");
listaFuncionarios.Remove("Oscar");

int indice = listaFuncionarios.IndexOf("Maria");
Console.WriteLine($"Índice da Maria = {indice}");

listaFuncionarios.Sort();
listaFuncionarios.Reverse();
```

Veja que os métodos `IndexOf()`, `Sort()` e `Reverse()` são nativos em listas (`List<T>`), permitindo **chamá-los diretamente** com `nomeDaLista.Metodo()`. Já nos arrays, esses métodos não fazem parte da estrutura, sendo acessados pela **classe auxiliar Array**, como em `Array.Metodo(nomeDoArray)`. Sem essa classe, seria preciso implementar manualmente essas operações. Esse é um dos benefícios das coleções genéricas: **seus métodos não dependem de classes utilitárias**.

## Conjuntos

Um outro tipo de coleção genérica bastante utilizada são os conjuntos, representados pela classe `HashSet`.

Os conjuntos são estruturas que **não permitem dados duplicados**. Além disso, assim como nas listas, a adição e remoção de elementos é bastante prática. Outra característica importante é que, em conjuntos, **não conseguimos manter a ordem de inserção** dos elementos.

Por ser uma coleção genérica, ao declarar um `HashSet`, devemos dizer qual o tipo de dados ele irá armazenar. Esse tipo é passado entre os sinais `<>`.

```
HashSet<string> setFuncionarios = new HashSet<string>
{
    "João", "Maria", "Cláudia", "Oscar"
};
```

## Conjuntos

Há duas formas de criar um conjunto. Assim como na lista, em nenhuma das inicializações o tamanho é fixo.

```
HashSet<string> setFuncionarios = new HashSet<string>
{
    "João", "Maria", "Cláudia", "Oscar"
};
```

```
HashSet<string> setFuncionarios = new HashSet<string>();

setFuncionarios.Add("João");
setFuncionarios.Add("Maria");
setFuncionarios.Add("Cláudia");
setFuncionarios.Add("Oscar");
```

# Conjuntos

Nos conjuntos, também temos os métodos `Add()` e `Remove()` de forma nativa. Inclusive, devido aos detalhes de implementação, esses métodos costumam ser mais rápidos em conjuntos do que em listas.

```
HashSet<string> setFuncionarios = new HashSet<string>
{
    "João", "Maria", "Cláudia", "Oscar"
};

setFuncionarios.Add("Joana");
setFuncionarios.Add("Maria");
setFuncionarios.Remove("João");
```

Como o elemento "Maria" já foi adicionado uma vez no conjunto, ele não será adicionado novamente. Dessa forma, o `Add("Maria")` é ignorado.

Como a ordem de inserção não é mantida nos hashsets, eles não guardam índices para os elementos. Assim, não faz sentido que eles tenham métodos como `Sort()`, `Reverse()` ou `IndexOf()`, que dependem de índices para funcionar.

# Conjuntos

Por outro lado, um dos métodos mais usado em conjuntos é o `Contains()`, que serve para verificar se um elemento existe ou não no conjunto. Essa também é uma operação mais rápida nos hashsets em comparação com lists.

```
if (setFuncionarios.Contains("Oscar"))
{
    Console.WriteLine("Oscar faz parte do conjunto");
}

string elemento = setFuncionarios[2];
```

Ao lado, temos o uso do método `Contains()`. Além disso, há uma tentativa de fazer uma indexação no conjunto, que é sinalizada como erro pelo compilador.

Mas por quê não guardar índices? Essa não é uma operação interessante em coleções? **Depende**.

Os `HashSets` se baseiam em **conjuntos matemáticos**. Nesses tipos de conjunto, não há repetições de elementos, e não há uma ordem específica para eles. A operação mais importante em um conjunto é saber se um elemento pertence ou não a ele. Dessa forma, **utilizamos essa estrutura quando nossos dados realmente podem ser modelados para ela**.

## Dictionary

A última coleção genérica que iremos explorar são os dicionários, representados pela classe `Dictionary`.

Utilizamos essa estrutura de dados quando trabalhamos com pares de **chave e valor**, **focando principalmente em buscas**. A seguir, podemos ver um exemplo de situação em que isso ocorre:

| ID      | Nome          |
|---------|---------------|
| 4587963 | Teclado       |
| 3374561 | Cadeira gamer |
| 2456987 | Notebook      |
| 6356984 | Teclado       |
| 8647921 | Mouse         |

Imagine que estamos trabalhando em uma loja de eletrônicos, e os produtos têm ids únicos, conforme os valores da tabela. Se os ids começassem em 0 e fossem aumentando sequencialmente, poderíamos representar os produtos com listas. Buscar o produto 4 seria acessar a lista na posição 4. Porém, nesse caso os ids aparecem aleatoriamente, sendo mais fácil armazená-los também.

## Dictionary

Para representar os produtos da tabela como um dicionário, iremos guardar duas informações de uma vez:

- **Chave (deve ser única) => ID do produto**
- **Valor => Nome do produto**

```
Dictionary<int, string> produtos = new Dictionary<int, string>
{
    { 4587963, "Teclado" },
    { 3374561, "Cadeira gamer" },
    { 2456987, "Notebook" },
    { 6356984, "Teclado" },
    { 8647921, "Mouse" }
};
```

# Dictionary

Há duas formas de inicializar dicionários:

```
Dictionary<int, string> produtos = new Dictionary<int, string>
{
    { 4587963, "Teclado" },
    { 3374561, "Cadeira gamer" },
    { 2456987, "Notebook" },
    { 6356984, "Teclado" },
    { 8647921, "Mouse" }
};
```

Agora, dentro dos generics (<>), iremos passar dois tipos: primeiro o tipo da chave a ser guardada, e depois o tipo do valor dos elementos. Nesse exemplo, a chave é do tipo int e o valor é do tipo string.

```
Dictionary<int, string> produtos = new Dictionary<int,
string>();
produtos.Add(4587963, "Teclado");
produtos.Add(3374561, "Cadeira gamer");
produtos.Add(2456987, "Notebook");
produtos.Add(6356984, "Teclado");
produtos.Add(8647921, "Mouse");
```

## Dictionary

A operação mais comum em dicionários é a **busca pela chave**. Essa busca pode ser feita com os colchetes: `nomeDicionario[chaveBuscada]`.

```
Dictionary<string, string> palavras = new Dictionary<string, string>
{
    {"olá", "hello" },
    { "mundo", "world"}
};

Console.WriteLine($"A tradução de olá é {palavras["olá"]});
```

Podemos interpretar a estrutura de dados `Dictionary` como um dicionário mesmo! Pense em um dicionário de tradução português-inglês: a chave é a palavra em português, que é única, e o valor é a tradução em inglês.

## Dictionary

Também podemos percorrer um dicionário. Essa operação é feita da seguinte forma:

```
foreach (KeyValuePair<int, string> produto in produtos)
{
    Console.WriteLine($"ID: {produto.Key} - Nome: {produto.Value}");
}

foreach (var produto in produtos)
{
    Console.WriteLine($"ID: {produto.Key} - Nome: {produto.Value}");
}
```

No foreach, a variável `produto` serve para representar um único elemento por vez. Como cada elemento é um par de chave e valor, esses elementos são então representados pelo tipo `KeyValuePair`, em que indicamos também o tipo da chave e do valor no generics (`<>`). Caso queiramos simplificar, basta usar o `var` para que haja a inferência de tipos.

## RESUMINDO - Quando usar cada coleção?

| Coleção                    | Quando usar  | Características   | Vantagens   | Desvantagens   |
|----------------------------|--|---|---|--|
| Array                      | Quando o tamanho é fixo e a busca por índice é frequente.                        | Estrutura de tamanho fixo, acessível por índice.                        | Mais rápido para acesso direto ao índice.                   | Não pode ser redimensionado, exige criação de novo array para expansão.        |
| List<T>                    | Quando precisamos armazenar dados ordenadamente e permitir alterações dinâmicas. | Estrutura dinâmica que mantém a ordem dos elementos.                    | Fácil de manipular, permite adição e remoção dinâmica.      | Pode ser menos eficiente para buscas em coleções muito grandes.                |
| HashSet<T>                 | Quando precisamos armazenar elementos únicos e a ordem de inserção não importa.  | Estrutura baseada em conjuntos matemáticos.                             | Busca, inserção e remoção rápidas e não permite duplicatas. | Não mantém a ordem dos elementos e não permite acesso por índice.              |
| Dictionary< TKey, TValue > | Quando precisamos mapear pares chave-valor para busca eficiente.                 | Estrutura de pares onde cada chave é única e acessa um valor associado. | Acesso rápido, ideal para dados indexados.                  | Chaves devem ser únicas, pode usar mais memória dependendo do volume de dados. |



Compartilhe um resumo de seus novos  
conhecimentos em suas redes sociais.  
[#aprendizadoalura](#)