# Estrutura de Dados em C#

## 1. Estrutura de Dados complexas em C#

O .NET Framework inclui vários tipos de dados internos, como int, decimal, string e Boolean. Esses tipos de dados podem ser identificados como tipos de dados simples porque consistem em um valor único e simples, como um número, valor de texto ou uma configuração verdadeira ou falsa. Você poderia argumentar que String não é um tipo de dados simples e, até certo ponto, está correto. C / C ++ e outras linguagens semelhantes consideram uma String como uma matriz de caracteres. Como este módulo cobrirá matrizes, podemos concordar com isso e concordamos. A principal diferença para o C # é que podemos usar um valor String como uma estrutura de dados simples sem nos preocupar em acessá-lo como uma matriz de caracteres, lidando com um ponteiro para o início da matriz ou qualquer outra função de string encontrada em C. , porque o C # é flexível o suficiente para permitir isso, mas para os tipos de dados em C #.

Os tipos de dados complexos são adequados para cenários em que você precisa armazenar vários itens em uma única entidade. Considere um baralho de cartas, os dias da semana, meses do ano e itens ainda mais complexos, como um objeto no código para representar um carro. Cada um desses exemplos requer vários valores para refletir o conceito. Pensamos em uma semana como uma entidade única que consiste em sete dias. Quando falamos de uma semana, fazemos isso com o conhecimento inerente ao que consiste uma semana.

## 2. Matrizes

Uma matriz é um conjunto de objetos que são agrupados e gerenciados como uma unidade. Você pode pensar em uma matriz como uma sequência de elementos, todos do mesmo tipo. Você pode criar matrizes simples que tenham uma dimensão (uma lista), duas dimensões (uma tabela), três dimensões (um cubo) e assim por diante. Matrizes no Visual C # têm os seguintes recursos:

- Cada elemento da matriz contém um valor.
- As matrizes são indexadas a zero, ou seja, o primeiro item da matriz é o elemento 0.
- O tamanho de uma matriz é o número total de elementos que ela pode conter.
- As matrizes podem ser unidimensionais, multidimensionais ou serrilhadas.
- A classificação de uma matriz é o número de dimensões na matriz.

Matrizes de um tipo específico podem conter apenas elementos desse tipo. Se você precisar manipular um conjunto de objetos ou tipos de valor diferentes, considere usar um dos tipos de coleção definidos namespace System.Collections.

### 2.1. Criando e usando matrizes de dimensões únicas

Ao declarar uma matriz, você especifica o tipo de dados que ela contém e um nome para a matriz. Declarar uma matriz coloca a matriz no escopo, mas na verdade não aloca nenhuma memória para ela. O CLR cria fisicamente a matriz quando você usa a nova palavra-chave. Neste ponto, você deve especificar o tamanho da matriz.

Para declarar uma matriz unidimensional, você especifica o tipo de elementos na matriz e usa colchetes, [] para indicar que uma variável é uma matriz. Posteriormente, você especifica o tamanho da matriz ao alocar memória para a matriz usando a nova palavra-chave. O tamanho de uma matriz pode ser qualquer expressão inteira. O exemplo de código a seguir mostra como criar uma matriz unidimensional de números inteiros com elementos de zero a nove.

```
int[] arrayName = new int[10];
```

### 2.2. Acessando dados em uma matriz

Você pode acessar dados em uma matriz de várias maneiras, como especificando o índice de um elemento específico necessário ou iterando por toda a matriz e retornando cada elemento em sequência.

O exemplo de código a seguir usa um índice para acessar o elemento no índice dois.

```
//Accessing Data by Index
int[] oldNumbers = { 1, 2, 3, 4, 5 };

//number will contain the value 3
int number = oldNumbers[2];
```

Você pode percorrer uma matriz usando um loop for. Você pode usar a propriedade Length da matriz para determinar quando parar o loop.

O exemplo de código a seguir mostra como usar um loop for para iterar através de uma matriz.

```
//Iterating Over an Array
int[] oldNumbers = { 1, 2, 3, 4, 5 };
for (int i = 0; i < oldNumbers.Length; i++)
    int number = oldNumbers[i];
```

### 2.3. Matrizes multidimensionais

Uma matriz pode ter mais de uma dimensão. O número de dimensões corresponde ao número de índices usados para identificar um elemento individual na matriz. Você pode especificar até 32 dimensões, mas raramente precisará de mais de três. Você declara uma variável de matriz multidimensional assim como declara uma matriz unidimensional, mas separa as dimensões usando vírgulas. O exemplo de código a seguir mostra como criar uma matriz de números inteiros com três dimensões.

```
// Create an array that is 10 long(rows) by 10 wide(columns)
int[ , ] arrayName = new int[10,10];
```

Para acessar elementos em uma matriz multidimensional, você deve incluir todos os índices, como no código de exemplo aqui.

```
// Access the element in the first row and first
column int value = arrayName[0,0];

//Access the element in the first row and second
column int value2 = arrayName[0, 1];

//Access the element in the second row and first
column int value2 = arrayName[1, 0];
```

### 2.4. Matrizes irregulares

Uma matriz irregular é simplesmente uma matriz de matrizes, e o tamanho de cada matriz pode variar. Matrizes irregulares são úteis para modelar estruturas de dados esparsas, nas quais nem sempre você deseja alocar memória para cada item, se não for utilizado. O exemplo de código a seguir mostra como declarar e inicializar uma matriz irregular. Observe que você deve especificar o tamanho da primeira matriz, mas não deve especificar o tamanho das matrizes contidas nessa matriz. Você aloca memória para cada matriz dentro de uma matriz irregular separadamente, usando a palavra-chave new.

```
int[][] jaggedArray = new int[10][];
jaggedArray[0] = new Type[5]; // Can specify different sizes.
jaggedArray[1] = new Type[7];
...
jaggedArray[9] = new Type[21];
```

**3. Enums**

Um tipo de enumeração, ou enum, é uma estrutura que permite criar uma variável com um conjunto fixo de valores possíveis. O exemplo mais comum é usar uma enumeração para definir o dia da semana. Existem apenas sete valores possíveis para os dias da semana e você pode estar razoavelmente certo de que esses valores nunca serão alterados.

Uma prática recomendada seria definir sua enum diretamente dentro de um namespace, para que todas as classes nesse namespace tenham acesso a ele, se necessário. Você também pode aninhar suas enumerações em classes ou estruturas.

Por padrão, os valores de enumeração começam em 0 e cada membro sucessivo é aumentado em um valor de 1.

Para criar uma enumeração, você a declara em seu arquivo de código com a seguinte sintaxe, que demonstra a criação de uma enumeração chamada Dia, que contém os dias da semana:

*enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };*

Por padrão, os valores de enumeração começam em 0 e cada membro sucessivo é aumentado em um valor de 1. Como resultado, a enumeração anterior 'Dia' conteria os valores:

- Domingo = 0
- Segunda = 1
- Terça-feira = 2
- etc.

Você pode alterar o padrão especificando um valor inicial para sua enumeração, como no exemplo a seguir.

*enum Day { Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };*

Neste exemplo, o domingo recebe o valor 1 em vez do padrão 0. Agora, segunda-feira é 2, terça-feira é 3, etc.

A palavra-chave enum é usada para especificar o "tipo" que a variável Day será. Nesse caso, um tipo de enumeração. As enums suportam tipos de dados intrínsecos e podem ser qualquer um dos seguintes:

- byte
- sbyte
- baixo
- ushort
- int
- uint
- longo
- Ulong

Para alterar o tipo de dados padrão da sua enumeração, você precede a lista com um tipo de dados da lista acima, como:

*enum Day : short { Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };*

O tipo subjacente especifica quanto armazenamento será alocado para cada enumerador na enumeração. Durante o tempo de compilação, sua enumeração será convertida em literais numéricos no seu código. Se você estiver usando o Visual Studio, o recurso Intellisense será

totalmente capaz de reconhecer suas enumerações e exibirá os valores de sequência automaticamente no IDE conforme você digita o nome da enumeração.

É importante observar que você precisará usar uma conversão explícita se desejar converter de um tipo de enumeração para um tipo integral. Considere este exemplo em que a instrução atribui o enumerador Sun a um tipo int, com uma conversão, para converter de enum em int.

*int x = (int)Days.Sun;*

### 3.1. Usando um Enum

Para usar a enum, você cria uma instância da sua variável enum e especifica qual membro da enum você deseja usar.

*Day favoriteDay = Day.Friday;*

O uso de enums possui várias vantagens em relação ao uso de texto ou tipos numéricos:

- Gerenciamento aprimorado. Ao restringir uma variável a um conjunto fixo de valores válidos, é menos provável que você experimente argumentos inválidos e erros de ortografia.
- Experiência aprimorada do desenvolvedor. No Visual Studio, o recurso IntelliSense solicitará os valores disponíveis quando você usar uma enumeração.
- Legibilidade de código aprimorada. A sintaxe enum facilita a leitura e a compreensão do seu código.

Cada membro de uma enumeração tem um nome e um valor. O nome é a sequência que você define entre chaves, como domingo ou segunda-feira. Por padrão, o valor é um número inteiro. Se você não especificar um valor para cada membro, os membros receberão valores incrementais iniciados com 0. Por exemplo, Day.Sunday é igual a 0 e Day.Monday é igual a 1.

O exemplo a seguir mostra como você pode usar nomes e valores de forma intercambiável:

### 3.2. Usando nomes e valores de enumeração de maneira intercambiável

```
// Set an enum variable by name.
Day favoriteDay = Day.Friday;

// Set an enum variable by value.
Day favoriteDay = (Day)4;
```

## 4. Structs

No Visual C #, uma estrutura é uma construção de programação que você pode usar para definir tipos personalizados. Estruturas são estruturas de dados essencialmente leves que representam informações relacionadas como um único item. Por exemplo:

- Uma estrutura chamada Point pode consistir em campos para representar uma coordenada x e uma coordenada y.
- Uma estrutura chamada Circle pode consistir em campos para representar uma coordenada x, uma coordenada y e um raio.
- Uma estrutura chamada Cor pode consistir em campos para representar um componente vermelho, um componente verde e um componente azul.

A maioria dos tipos internos do Visual C #, como int, bool e char, são definidos por estruturas. Você pode usar estruturas para criar seus próprios tipos que se comportam como tipos internos.

### 4.1. Criando uma Struct

Você usa a palavra-chave struct para declarar uma estrutura, conforme mostrado no exemplo a seguir:

```
//Declaring a Struct
public struct Coffee
{
public int Strength;
public string Bean;
public string CountryOfOrigin;
// Other methods, fields, properties, and events.
}
```

A palavra-chave struct é precedida por um modificador de acesso - público no exemplo acima - que especifica onde você pode usar o tipo. Você pode usar os seguintes modificadores de acesso em suas declarações struct:

Detalhes do modificador de acesso:

- Public: O tipo está disponível para código em execução em qualquer assembly.
- Internal: O tipo está disponível para qualquer código dentro do mesmo assembly, mas não está disponível para codificar em outro assembly. Este é o valor padrão se você não especificar um modificador de acesso.
- Private: O tipo está disponível apenas para código dentro da estrutura que o contém. Você só pode usar o modificador de acesso privado com estruturas aninhadas.

As estruturas podem conter uma variedade de membros, incluindo construtores, campos, constantes, propriedades, indexadores, métodos, operadores, eventos e até tipos aninhados. Lembre-se de que as estruturas devem ser leves, portanto, se você adicionar vários métodos, construtores e eventos, considere usar uma classe.

### 4.2. Usando uma Struct

Para criar uma instância de uma estrutura, use a palavra-chave new, conforme mostrado no exemplo a seguir:

```
Instantiating a Struct
Coffee coffee1 = new Coffee();
coffee1.Strength = 3;
coffee1.Bean = "Arabica";
```

*coffee1.CountryOfOrigin = "Kenya";*

### 4.3. Inicializando estruturas

Você deve ter notado que a sintaxe para instanciar uma estrutura, por exemplo, new Coffee(), é semelhante à sintaxe para chamar um método. Isso ocorre quando você instancia uma estrutura, na verdade, está chamando um tipo especial de método chamado construtor. Um construtor é um método na estrutura que tem o mesmo nome que a estrutura. Ao instanciar uma estrutura sem argumentos, como o novo Coffee(), você está chamando o construtor padrão criado pelo compilador do Visual C #. Se você desejar especificar valores de campo padrão ao instanciar uma estrutura, poderá adicionar construtores que aceitem parâmetros à sua estrutura. O exemplo a seguir mostra como criar um construtor em uma estrutura:

```
public struct Coffee
{
// This is the custom constructor.
public Coffee(int strength, string bean, string countryOfOrigin)

// These statements declare the struct fields and set the default values.
public int Strength;
public string Bean;
public string CountryOfOrigin;
// Other methods, fields, properties, and events.
}

Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
```

Você pode adicionar vários construtores à sua estrutura, com cada construtor aceitando uma combinação diferente de parâmetros. No entanto, você não pode adicionar um construtor padrão a uma estrutura porque ele é criado pelo compilador.

## 5. Árvores

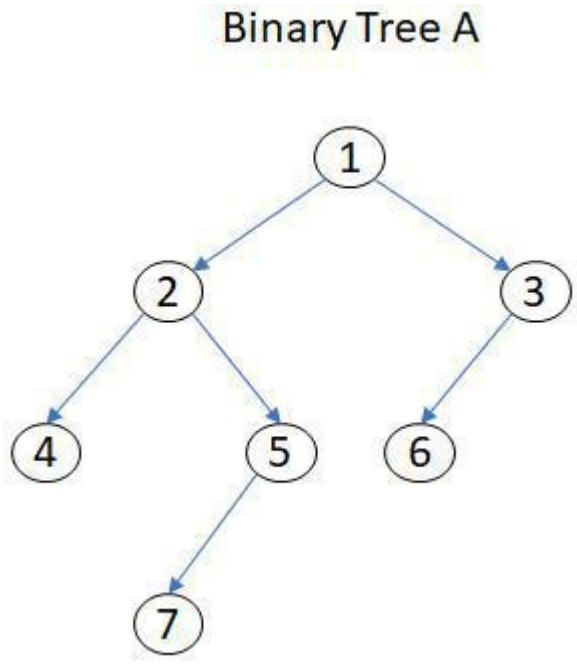
Neste tópico, examinaremos a árvore binária. Como veremos, as árvores binárias armazenam dados de maneira não linear. Se você já olhou para uma tabela de genealogia ou para a cadeia de comando de uma corporação, viu dados organizados em uma árvore. Uma árvore é composta por uma coleção de nós, em que cada nó tem alguns dados associados e um conjunto de filhos. Os filhos de um nó são aqueles que aparecem imediatamente abaixo do próprio nó. O pai de um nó é o nó imediatamente acima dele. A raiz de uma árvore é o nó único que não contém pai.

Todas as árvores exibem as seguintes propriedades:

- Existe precisamente uma raiz.
- Todos os nós, exceto a raiz, têm exatamente um pai.
- Não há ciclos. Ou seja, iniciando em um nó específico, não há um caminho que possa levá-lo de volta ao nó inicial. As duas primeiras propriedades - que existe uma raiz e que todos os nós salvam a raiz têm um pai - garantem a inexistência de ciclos.

As árvores são úteis para organizar dados em uma hierarquia. Como discutiremos mais adiante neste artigo, o tempo para procurar um item pode ser drasticamente reduzido, organizando de forma inteligente a hierarquia. Antes de chegarmos a esse tópico, precisamos discutir primeiro um tipo especial de árvore, a árvore binária.

Uma árvore binária é um tipo especial de árvore, em que todos os nós têm no máximo dois filhos. Para um determinado nó em uma árvore binária, o primeiro filho é referido como filho esquerdo, enquanto o segundo filho é referido como filho direito. A figura abaixo mostra uma árvore binária.




A árvore binária A possui 7 nós, com o nó 1 como raiz. O filho esquerdo do nó 1 é o nó 2; o filho direito do nó 1 é o nó 3. Observe que um nó não precisa ter um filho esquerdo e um filho direito. Na árvore binária A, o nó 5, por exemplo, possui apenas um filho esquerdo como o nó 3. Além disso, um nó não pode ter filhos. Na árvore binária b, os nós 4, 6 e 7 não têm filhos.

Os nós que não têm filhos são chamados de nós folha. Os nós que têm um ou dois filhos são chamados de nós internos. Usando essas novas definições, os nós folha na árvore binária A são os nós 4, 6 e 7; os nós internos são os nós 1, 2, 3 e 5.

## 6. Coleções

Quando você cria vários itens do mesmo tipo, independentemente de serem números inteiros, seqüências de caracteres ou um tipo personalizado como Coffee , você precisa de uma maneira de gerenciar os itens como um conjunto. Você precisa contar o número de itens no conjunto, adicionar ou remover itens do conjunto e iterar pelo item, um por vez. Para fazer isso, você pode usar uma coleção .

As coleções são uma ferramenta essencial para gerenciar vários itens. Eles também são essenciais para o desenvolvimento de aplicativos gráficos. Controles como caixas de listagem e menus suspensos geralmente são vinculados a dados para coleções.

O namespace System.Collections fornece uma variedade de coleções de uso geral que inclui listas, dicionários, filas e pilhas. A tabela a seguir mostra as classes de coleção mais importantes no espaço para nome System.Collections:

| Class | Description |
|---|---|
| ArrayList | The ArrayList is a general-purpose list that stores a linear collection of objects. The ArrayList includes methods and properties that enable you to add items, remove items, count the number of items in the collection, and sort the collection. |
| BitArray | The BitArray is a list class that represents a collection of bits as Boolean values. The BitArray is most commonly used for bitwise operations and Boolean arithmetic, and includes methods to perform common Boolean operations such as AND, NOT, and XOR. |
| Hashtable | The Hashtable class is a general-purpose dictionary class that stores a collection of key/value pairs. The Hashtable includes methods and properties that enable you to retrieve items by key, add items, remove items, and check for particular keys and values within the collection. |
| Queue | The Queue class is a first in, first out collection of objects. The Queue includes methods to add objects to the back of the queue (Enqueue) and retrieve objects from the front of the queue (Dequeue). |
| SortedList | The SortedList class stores a collection of key/value pairs that are sorted by key. In addition to the functionality provided by the Hashtable class, the SortedList enables you to retrieve items either by key or by index. |
| Stack | The Stack class is a first in, last out or last in, first out (LIFO) collection of objects. The Stack includes methods to view the top item in the collection without removing it (Peek), add an item to the top of the stack (Push), and remove and return the item at the top of the stack (Pop). |

### 6.1. Escolhendo coleções

Todas as classes de coleção compartilham várias características comuns. Para gerenciar uma coleção de itens, você deve ser capaz de:

- Adicione itens à coleção.
- Remova os itens da coleção.
- Recupere itens específicos da coleção.
- Conte o número de itens na coleção.
- Repita os itens da coleção, um item de cada vez.

Cada classe de coleção no Visual C # fornece métodos e propriedades que suportam essas operações principais. Além dessas operações, no entanto, você desejará gerenciar coleções de maneiras diferentes, dependendo dos requisitos específicos do seu aplicativo. As classes de coleção no Visual C # se enquadram nas seguintes categorias amplas:

- As classes de lista armazenam coleções lineares de itens. Você pode pensar em uma classe de lista como uma matriz unidimensional que se expande dinamicamente à medida que você adiciona itens. Por exemplo, você pode usar uma classe de lista para manter uma lista de bebidas disponíveis em sua cafeteria.
- As classes de dicionário armazenam uma coleção de pares de chave / valor. Cada item da coleção consiste em dois objetos - a chave e o valor. O valor é o objeto que você deseja armazenar e recuperar, e a chave é o objeto que você usa para indexar e procurar o valor. Na maioria das classes de dicionário, a chave deve ser única, enquanto valores duplicados são perfeitamente aceitáveis. Por exemplo, você pode usar uma classe de dicionário para manter

uma lista de receitas de café. A chave conteria o nome exclusivo do café e o valor conteria os ingredientes e as instruções para fazer o café.

- As classes de fila representam uma coleção de objetos primeiro a entrar, primeiro a sair. Os itens são recuperados da coleção na mesma ordem em que foram adicionados. Por exemplo, você pode usar uma classe de fila para processar pedidos em uma cafeteria para garantir que os clientes recebam suas bebidas por sua vez.
- As classes de pilha representam uma última coleção de objetos, primeiro a entrar. O item que você adicionou à coleção pela última vez é o primeiro item que você recupera. Por exemplo, você pode usar uma classe de pilha para determinar os 10 visitantes mais recentes à sua cafeteria.

### 6.2. Usando coleções

A coleção de listas mais usada é a classe ArrayList. O ArrayList armazena itens como uma coleção linear de objetos. Você pode adicionar objetos de qualquer tipo a uma coleção ArrayList, mas o ArrayList representa cada item da coleção como uma instância System.Object. Quando você adiciona um item a uma coleção ArrayList, o ArrayList lança implicitamente ou converte seu item no tipo de Objeto. Ao recuperar itens da coleção, você deve explicitamente converter o objeto de volta ao seu tipo original.

O exemplo a seguir mostra como adicionar e recuperar itens de uma coleção ArrayList:

```
// Create a new ArrayList collection.
ArrayList beverages = new ArrayList();

// Create some items to add to the collection.
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
Coffee coffee2 = new Coffee(3, "Arabica", "Vietnam");
Coffee coffee3 = new Coffee(4, "Robusta", "Indonesia");

// Add the items to the collection.
// Items are implicitly cast to the Object type when you add them.
beverages.Add(coffee1);
beverages.Add(coffee2);
beverages.Add(coffee3);

// Retrieve items from the collection.
// Items must be explicitly cast back to their original type.
Coffee firstCoffee = (Coffee)beverages[0];
Coffee secondCoffee = (Coffee)beverages[1];
```

Ao trabalhar com coleções, uma das tarefas de programação mais comuns será a iteração sobre a coleção. Essencialmente, isso significa que você recupera cada item da coleção por sua vez, geralmente para renderizar uma lista de itens, avaliar cada item com base em alguns critérios ou extrair valores específicos de membros de cada item. Para iterar sobre uma coleção, use um loop foreach . O loop foreach expõe cada item da coleção, por sua vez, usando o nome da variável que você especifica na declaração do loop.

O exemplo a seguir mostra como iterar sobre uma coleção ArrayList:

```
// Iterating Over a List Collection
foreach(Coffee coffee in beverages)
{
   Console.WriteLine("Bean type: {0}", coffee.Bean);
   Console.WriteLine("Country of origin: {0}", coffee.CountryOfOrigin);
   Console.WriteLine("Strength (1-5): {0}", coffee.Strength);
}
```

As classes de dicionário armazenam coleções de pares de chave / valor. A classe de dicionário mais usada é o Hashtable. Ao adicionar um item a uma coleção Hashtable, você deve especificar uma chave e um valor. Tanto a chave como o valor podem ser instâncias de qualquer tipo, mas o Hashtable lança implicitamente a chave e o valor no tipo de objeto.

O exemplo a seguir mostra como adicionar e recuperar itens de uma coleção Hashtable. Nesse caso, a chave e o valor são cadeias de caracteres:

```
// Create a new Hashtable collection.
Hashtable ingredients = new Hashtable();

// Add some key/value pairs to the collection.
ingredients.Add("Café au Lait", "Coffee, Milk");
ingredients.Add("Café Mocha", "Coffee, Milk, Chocolate");
ingredients.Add("Cappuccino", "Coffee, Milk, Foam");
ingredients.Add("Irish Coffee", "Coffee, Whiskey, Cream, Sugar");
ingredients.Add("Macchiato", "Coffee, Milk, Foam");

// Check whether a key exists.
if(ingredients.ContainsKey("Café Mocha"))
{
  // Retrieve the value associated with a key.
  Console.WriteLine("The ingredients of a Café Mocha are: {0}", ingredients["Café Mocha"]);
}
```

As classes de dicionário, como o Hashtable, na verdade contêm duas coleções enumeráveis - as chaves e os valores. Você pode iterar sobre qualquer uma dessas coleções. Na maioria dos cenários, no entanto, é provável que você repita a coleção de chaves, por exemplo, para recuperar o valor associado a cada chave.

O exemplo a seguir mostra como iterar sobre as chaves em uma coleção Hashtable e recuperar o valor associado a cada chave:

```
// Iterating Over a Dictionary Collection
foreach(string key in ingredients.Keys)
{
  // For each key in turn, retrieve the value associated with the key.
  Console.WriteLine("The ingredients of a {0} are {1}", key, ingredients[key]);
}
```

### 6.3. Query Collections

Algumas coleções no .NET Framework não oferecem suporte ao uso de notação de matriz para acessar itens na coleção. Essas coleções fornecem o método Find para localizar itens na coleção. O método Find requer que um predicado seja usado como critério para sua pesquisa. Nesse caso, o predicado se torna um método que examinará cada item da coleção, retornando um valor booleano com base nos resultados da correspondência. A pesquisa termina quando um item é encontrado.

Predicados são tipicamente expressos na forma de uma expressão lambda. Expressões lambda são uma expressão em C # que retorna um método. Semelhante aos métodos com os quais você

já está familiarizado, uma expressão lambda contém uma lista de parâmetros e um corpo de método, mas não contém um nome de método nem um tipo de retorno. O tipo de retorno é inferido a partir do contexto em que a expressão lambda é usada.

```
List<Employee> employees= new List<Employee>()
{
new Employee() { empID = 001, Name = "Tom", Department= "Sales"},
new Employee() { empID = 024, Name = "Joan", Department= "HR"},
new Employee() { empID = 023, Name = "Fred", Department= "Accounting" },
new Employee() { empID = 040, Name = "Mike", Department= "Sales" }, };

// Find the member of the list that has an employee id of 023
Employee match = employees.Find((Employee p) => { return p.empID == 023; });
Console.WriteLine("empID: {0}\nName: {1}\nDepartment: {2}", match.empID, match.Name,
match.Department);
```

## 7. Generics

No tópico sobre coleções, você viu que o uso da classe ArrayList da coleção permitia armazenar diferentes tipos de dados na coleção. Isso contrasta com uma matriz em que os tipos de dados na matriz devem ser do mesmo tipo. No entanto, há um problema inerente ao ArrayList. O tópico discutiu que tudo o que você armazena no ArrayList é automaticamente convertido em um tipo de dados Object, o tipo raiz no .NET. Lembre-se dos tópicos de POO que o polimorfismo permite que uma classe base represente subclasses. O .NET usa Object como a classe base para todos os outros tipos de classe criados em C #.

O fato de o ArrayList armazenar todos os itens como Objeto também significa que, ao recuperar os itens, você precisa fazer alguma conversão ou conversão própria para garantir que os objetos retornados sejam convertidos de volta ao tipo que eram. Isso pode ser problemático e propenso a erros. Para ajudar a resolver esse problema, você deve estar usando generics.

Os generics permitem criar e usar coleções fortemente tipadas, que são seguras para o tipo, não exigem a conversão de itens e não exigem que você marque e desmarque tipos de valor.

### 7.1. Vantagens dos generics

Considere um exemplo em que você usa um ArrayList para armazenar uma coleção de objetos Coffee. Você pode adicionar objetos de qualquer tipo a um ArrayList. Suponha que um desenvolvedor adicione um objeto do tipo Tea à coleção. O código será construído sem reclamação. No entanto, uma exceção de tempo de execução ocorrerá se o método Sort for chamado, porque a coleção não pode comparar objetos de tipos diferentes. Além disso, quando você recupera um objeto da coleção, deve convertê-lo no tipo correto. Se você tentar converter o objeto para o tipo errado, ocorrerá uma exceção de tempo de execução de conversão inválida.

O exemplo a seguir mostra as limitações de segurança de tipo da abordagem ArrayList:

```
// Type Safety Limitations for Non-Generic Collections
var coffee1 = new Coffee();
var coffee2 = new Coffee();
var tea1 = new Tea();
var arrayList1 = new ArrayList();
arrayList1.Add(coffee1);
arrayList1.Add(coffee2);
arrayList1.Add(tea1);
// The Sort method throws a runtime exception because the collection is not homogenous.
arrayList1.Sort();
// The cast throws a runtime exception because you cannot cast a Tea instance to a Coffee instance.
Coffee coffee3 = (Coffee)arrayList1[2];
```

Como alternativa ao ArrayList, suponha que você use uma generic List<T> para armazenar uma coleção de objetos Coffee. Ao instanciar a lista, você fornece um argumento de tipo Coffee. Nesse caso, sua lista é garantida como homogênea, porque seu código não será criado se você tentar adicionar um objeto de qualquer outro tipo. O método Sort funcionará porque sua coleção é homogênea. Por fim, o indexador retorna objetos do tipo Coffee, em vez de System.Object, portanto, não há risco de exceções de conversão inválidas.

O exemplo a seguir mostra uma alternativa à abordagem ArrayList usando a classe generic List<T>:

```
// Type Safety in Generic Collections
var coffee1 = new Coffee();
var coffee2 = new Coffee();
var tea1 = new Tea();
var genericList1 = new List<Coffee>();
genericList1.Add(coffee1);
genericList1.Add(coffee2);
// This line causes a build error, as the argument is not of type Coffee.
genericList1.Add(tea1);
// The Sort method will work because the collection is guaranteed to be homogenous.
genericList1.Sort();
// The indexer returns objects of type Coffee, so there is no need to cast the return value.
Coffee coffee3 = genericList[1];
```

### 7.2. No Casting

Casting é um processo computacionalmente caro. Quando você adiciona itens a um ArrayList, seus itens são convertidos implicitamente no tipo System.Object. Ao recuperar itens de um ArrayList, você deve explicitamente convertê-los de volta ao tipo original. O uso de generics para adicionar e recuperar itens sem conversão melhora o desempenho do seu aplicativo.

### 7.3. Sem Boxing e Unboxing

Se você deseja armazenar tipos de valor em um ArrayList, os itens devem ser colocados na caixa quando forem adicionados à coleção e fora da caixa quando forem recuperados. O boxe e o unboxing incorrem em um grande custo computacional e podem retardar significativamente seus aplicativos, especialmente quando você itera sobre grandes coleções. Por outro lado, você pode adicionar tipos de valor a listas genéricas sem encaixar e desmarcar o valor.
O exemplo a seguir mostra a diferença entre coleções genéricas e não genéricas em relação a boxe e unboxing:

```
// Boxing and Unboxing: Generic vs. Non-Generic Collections
int number1 = 1;
var arrayList1 = new ArrayList();
// This statement boxes the Int32 value as a System.Object.
arrayList1.Add(number1);
// This statement unboxes the Int32 value.
int number2 = (int)arrayList1[0];
var genericList1 = new List<Int32>();
//This statement adds an Int32 value without boxing.
genericList1.Add(number1);
//This statement retrieves the Int32 value without unboxing.
int number3 = genericList1[0];
```

### 7.4. Criando e usando Generic Classes

Classes genéricas funcionam incluindo um parâmetro de tipo, T, na declaração de classe ou interface. Você não precisa especificar o tipo de T até instanciar a classe. Para criar uma classe genérica, você precisa:

- Adicionar o parâmetro de tipo T entre colchetes angulares após o nome da classe.
- Usar o parâmetro de tipo T no lugar dos nomes de tipo em seus membros.

O exemplo a seguir mostra como criar uma classe genérica:

```
// Creating a Generic Class
public class CustomList<T>
{
  public T this[int index] { get; set; }
  public void Add(T item)
  {
    // Method logic goes here.
  }
  public void Remove(T item)
  {
    // Method logic goes here.
  }
}
```

Ao criar uma instância da sua classe genérica, você especifica o tipo que deseja fornecer como um parâmetro de tipo. Por exemplo, se você quiser usar sua lista personalizada para armazenar objetos do tipo Coffee, você forneceria Coffee como o parâmetro type.

O exemplo a seguir mostra como instanciar uma classe genérica:

```
//Instantiating a Generic Class
CustomList<Coffee> clc = new CustomList<Coffee>;
Coffee coffee1 = new Coffee();
Coffee coffee2 = new Coffee();
clc.Add(coffee1);
clc.Add(coffee2);
Coffee firstCoffee = clc[0];
```

Quando você instancia uma classe, todas as instâncias de T da classe são efetivamente substituídas pelo parâmetro de tipo que você fornece. Por exemplo, se você instanciar a classe CustomList com um parâmetro de tipo Coffee:

- O método Add aceitará apenas um argumento do tipo Coffee.
- O método Remove aceita apenas um argumento do tipo Coffee.
- O indexador sempre fornecerá um valor de retorno do tipo Coffee.

### 7.5. Restringindo generics

Em alguns casos, pode ser necessário restringir os tipos que os desenvolvedores podem fornecer como argumentos quando instanciam sua classe genérica. A natureza dessas restrições dependerá da lógica que você implementa na sua classe genérica. Por exemplo, se uma classe de coleção usar uma propriedade chamada AverageRating para classificar os itens em uma coleção, você precisará restringir o parâmetro type às classes que incluem a propriedade AverageRating. Suponha que a propriedade AverageRating seja definida pela interface IBeverage. Para implementar essa restrição, você restringiria o parâmetro type às classes que implementam a interface IBeverage usando a palavra-chave where.

O exemplo a seguir mostra como restringir um parâmetro de tipo a classes que implementam uma interface específica:

```
// Constraining Type Parameters by Interface
public class CustomList<T> where T : IBeverage
{
}
```

Você pode aplicar os seis tipos de restrições a seguir nos parâmetros de tipo:

| Constraint | Description |
|---|---|
| where T : <name of interface> | The type argument must be, or implement, the specified interface. |
| where T : <name of base class> | The type argument must be, or derive from, the specified class |
| where T : U | The type argument must be, or derive from, the supplied type argument U |
| where T : new() | The type argument must have a public default constructor |
| where T : struct | The type argument must be a value type |
| where T : class | The type argument must be a reference type |

Você pode aplicar os seis tipos de restrições a seguir nos parâmetros de tipo:

```
// Apply Multiple Type Constraints
public class CustomList<T> where T : IBeverage, IComparable<T>, new()
{
}
```

### 7.6. Usando Generic List Collections

Um dos usos mais comuns e importantes dos generics é nas classes de coleção. As coleções genéricas se enquadram em duas grandes categorias: coleções genéricas de listas e coleções genéricas de dicionário. Uma lista genérica armazena uma coleção de objetos do tipo T.

A classe List<T> fornece uma alternativa fortemente tipada para a classe ArrayList. Como a classe ArrayList, a classe List<T> inclui métodos para:

- Adicionar um item
- Remover um item.
- Inserir um item em um índice especificado.
- Classificar os itens na coleção usando o comparador padrão ou um comparador especificado.
- Reordenar toda ou parte da coleção.

O exemplo a seguir mostra como usar a classe List<T>.

```
// Using the List<T> Class
string s1 = "Latte";
string s2 = "Espresso";
string s3 = "Americano";
string s4 = "Cappuccino";
string s5 = "Mocha";
// Add the items to a strongly-typed collection.
var coffeeBeverages = new List<String>();
coffeeBeverages.Add(s1);
coffeeBeverages.Add(s2);
```

```
coffeeBeverages.Add(s3);
coffeeBeverages.Add(s4);
coffeeBeverages.Add(s5);
// Sort the items using the default comparer.
// For objects of type String, the default comparer sorts the items alphabetically.
coffeeBeverages.Sort();
// Write the collection to a console window.
foreach(String coffeeBeverage in coffeeBeverages)
{
  Console.WriteLine(coffeeBeverage);
}
```

O namespace System.Collections.Generic também inclui várias coleções genéricas que fornecem funcionalidades mais especializadas:

- A classe LinkedList<T> fornece uma coleção genérica na qual cada item está vinculado ao item anterior na coleção e ao próximo item na coleção. Cada item da coleção é representado por um objeto LinkedListNode<T>, que contém um valor do tipo T, uma referência à instância pai LinkedList<T>, uma referência ao item anterior na coleção e uma referência ao próximo item da coleção.
- A classe Queue<T> representa uma coleção de objetos fortemente tipada, primeiro a entrar, primeiro a sair.
- A classe Stack<T> representa uma coleção de objetos fortemente tipada, último a entrar, primeiro a sair.

### 7.7. Usando Generic Dictionary Collections

As classes de dicionário armazenam coleções de pares de chave-valor. O valor é o objeto que você deseja armazenar e a chave é o objeto que você usa para indexar e recuperar o valor. Por exemplo, você pode usar uma classe de dicionário para armazenar receitas de café, onde a chave é o nome do café e o valor é a receita desse café. No caso de dicionários genéricos, a chave e o valor são fortemente tipados.

A classe Dictionary <TKey, TValue> fornece uma classe de dicionário de uso geral e com tipagem forte. Você pode adicionar valores duplicados à coleção, mas as chaves devem ser exclusivas. A classe lançará uma ArgumentException se você tentar adicionar uma chave que já existe no dicionário.

O exemplo a seguir mostra como usar a classe Dictionary <TKey, TValue>:

```
// Using the Dictionary<TKey, TValue> Class
// Create a new dictionary of strings with string keys.
var coffeeCodes = new Dictionary<String, String>();
// Add some entries to the dictionary.
coffeeCodes.Add("CAL", "Café Au Lait");
coffeeCodes.Add("CSM", "Cinammon Spice Mocha");
coffeeCodes.Add("ER", "Espresso Romano");
coffeeCodes.Add("RM", "Raspberry Mocha");
coffeeCodes.Add("IC", "Iced Coffee");
// This statement would result in an ArgumentException because the key already exists.
// coffeeCodes.Add("IC", "Instant Coffee");
// To retrieve the value associated with a key, you can use the indexer.
// This will throw a KeyNotFoundException if the key does not exist.
```

```
Console.WriteLine("The value associated with the key \"CAL\" is {0}", coffeeCodes["CAL"]);
// Alternatively, you can use the TryGetValue method.
// This returns true if the key exists and false if the key does not exist.
string csmValue = "";
if(coffeeCodes.TryGetValue("CSM", out csmValue))
{
   Console.WriteLine("The value associated with the key \"CSM\" is {0}", csmValue);
}
else
{
   Console.WriteLine("The key \"CSM\" was not found");
}
// You can also use the indexer to change the value associated with a key.
coffeeCodes["IC"] = "Instant Coffee";
```

As classes SortedList<TKey, TValue> e SortedDictionary<TKey, TValue> fornecem dicionários genéricos nos quais as entradas são classificadas por chave. A diferença entre essas classes está na implementação subjacente:

- A classe genérica SortedList usa menos memória que a classe genérica SortedDictionary.
- A classe SortedDictionary é mais rápida e eficiente na inserção e remoção de dados não classificados.