

Fundamentos de C#

Coleções de Dados



Tópicos Abordados



- Coleções de dados
 - Legadas
 - Com suporte a generics
- Listas
- Conjuntos
- Dicionários
- Filas
- Pilhas

Motivação



- Frequentemente, uma aplicação precisa agrupar dados de determinado tipo
 - Ex: lista de itens, fila de documentos
- Arrays podem ser usados, mas são limitados
 - Arrays têm poucos recursos e algumas deficiências
- Usar um conjunto bem definido de classes e interfaces que representam vários tipos de coleções de dados é mais interessante

Coleções de Dados



- A plataforma .NET fornece uma série de tipos de coleções dados
 - Estão disponíveis para a linguagem C#
- Alguns tipos
 - Lists (listas)
 - Sets (conjuntos)
 - Dictionaries (dicionários)
 - Queues (filas)
 - Stacks (pilhas)
- Além dessas, existem outras coleções mais especializadas

Coleções Legadas



- Antes do .NET 2.0, as coleções não usavam generics
- Namespace *System.Collections*
- Alguns exemplos
 - *ArrayList*, *SortedList*, *HashTable*
- Estas coleções podem estar presentes em códigos já existentes
- Não devem mais ser utilizadas em novos projetos
 - Trabalhar com coleções de dados sem o uso de generics traz uma série de desvantagens

Coleções com Suporte ao Generics



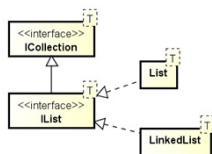
- As coleções de dados que você deve escolher devem ser aquelas que suportam o uso do generics
- Namespace **System.Collections.Generic**
- Interfaces importantes

Interface	Descrição
<i>ICollection<T></i>	Representa o conceito de coleção. <i>Count</i> , <i>Add()</i> , <i>Remove()</i> e <i>Clear()</i> são elementos importantes.
<i>IList<T></i>	Representa uma lista sequencial de elementos.
<i>ISet<T></i>	Representa um conjunto, onde não há duplicação de elementos.
<i>IDictionary<K, V></i>	Representa um dicionário, que armazena pares de chave e valor. Um valor pode ser obtido com base na sua chave.

Listas



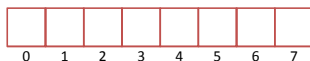
- Armazenam elementos de forma sequencial



A Classe `List<T>`



- Os elementos são armazenados em um array
- Cada elemento tem um índice



```
List<string> lista = new List<string>();  
lista.Add("C#");  
lista.Add("C++");  
lista.Add("Java");  
  
int count = lista.Count;
```

Cria uma lista vazia

Add() adiciona elementos na lista

Count conta o número de elementos da lista

Collection Initializers



- É uma sintaxe que permite a inserção de elementos no momento da criação da lista
 - Recurso semelhante ao dos arrays
 - Só pode ser usado no momento da criação da lista

```
List<string> lista = new List<string>() { "C#", "C++", "Java" };
```

Collection initializer

Na compilação, esta sintaxe é transformada em chamadas ao método Add()

Adicionando Elementos na Lista



- O método **Add()** adiciona um elemento no final da lista

```
lista.Add("C#");
```

- O método **AddRange()** adiciona vários elementos no final da lista

```
lista.AddRange(new string[] { "C#", "C++", "Java" });
```

- Os elementos também podem ser passados no construtor

```
List<string> lista =  
    new List<string>(new string[] { "C#", "C++", "Java" });
```

Adicionando Elementos na Lista



- O método **Insert()** adiciona um elemento no meio da lista
 - É necessário fornecer o índice

```
lista.Insert(1, "Lisp");
```

- O método **InsertRange()** adiciona vários elementos no meio da lista

```
lista.InsertRange(0, new string[] { "Lisp", "Prolog" });
```

Acessando Elementos na Lista




- Os elementos da uma lista podem ser acessados através do seu índice
 - O acesso é feito igual aos arrays

```
string primeiro = lista[0];  
string segundo = lista[1];
```

- A iteração sobre os elementos da lista também é possível

```
foreach (string e in lista)  
{  
    //...  
}  
  
for (int i = 0; i < lista.Count; i++)  
{  
    string e = lista[i];  
    //...  
}
```

Procurando Elementos na Lista




- A busca de um elemento na lista pode ser feita através dos métodos **IndexOf()** e **LastIndexOf()**

```
int index = lista.IndexOf("C++");
```

Retorna o índice do elemento, ou -1 se ele não for encontrado

- IndexOf()** também permite procurar um índice em apenas parte da lista
- LastIndexOf()** funciona da mesma forma, mas retorna o último índice encontrado

Procurando Elementos na Lista




- O método **Contains()** permite checar se um elemento existe na lista

```
bool b = lista.Contains("C#");
```

Retorna um booleano

Elementos iguais e diferentes



- Quando o dado é do tipo *string*, *int*, *char*, etc. é fácil determinar a igualdade
- E se o tipo for uma classe ou estrutura criada por você?
 - É preciso “ensinar” o C# a reconhecer a igualdade entre os objetos

```
class Linguagem  
{  
    public string Nome { get; set; }  
}
```

O Método *Equals()*



- A classe *object* define o método **Equals()**, que pode ser sobrescrito

```
public override bool Equals(object obj)
{
    if (obj == null)
    {
        return false;
    }

    Linguagem l = obj as Linguagem;

    if (l == null)
    {
        return false;
    }

    return this.Nome == l.Nome;
}
```

Neste exemplo, duas linguagens são iguais se possuem o mesmo nome

A Interface *IEquatable<T>*



- O método *Equals()* sempre recebe como parâmetro um tipo *object*
 - Isto pode causar problemas de performance no uso de *value types* (boxing e unboxing)
- Implementar a interface **IEquatable<T>** é a forma recomendada

```
class Linguagem : IEquatable<Linguagem>
{
    public bool Equals(Linguagem other)
    {
        return this.Nome == other.Nome;
    }
}
```

O parâmetro já é do tipo especificado

A Interface *IEquatable<T>*



- Se a interface *IEquatable<T>* é implementada, o *Equals()* de *object* também deve ser sobrescrito

```
class Linguagem : IEquatable<Linguagem>
{
    public string Nome { get; set; }

    public override bool Equals(object obj)
    {
        return Equals(obj as Linguagem);
    }

    public bool Equals(Linguagem other)
    {
        return this.Nome == other.Nome;
    }
}
```

Chama o *Equals()* de *IEquatable<T>*

Critério de Igualdade não Definido



- O que aconteceria se a classe não implementasse *IEquatable<T>* ou não sobrescrevesse o método *Equals()*?
- A implementação padrão do *Equals()* é utilizada
- Ela compara os valores de *value types* e as referências de *reference types*
 - Dois *value types* são iguais se possuem o mesmo tipo e valor
 - Dois *reference types* são iguais caso sejam o mesmo objeto na memória

Removendo Elementos da Lista



- A remoção pode ser feita por índice ou por elemento
 - Métodos **RemoveAt()** e **Remove()**

```
lista.RemoveAt(0);
```

Remove o primeiro elemento da lista

```
bool b = lista.Remove(linguagem);
```

Remove o elemento

Retorna um booleano indicando se a remoção foi bem sucedida

Ordenando a Lista



- A ordenação dos elementos da lista pode ser feita com o método **Sort()**
- Para que a ordenação funcione, o tipo deve implementar **IComparable<T>**
- Outra forma é usar um **IComparer<T>**

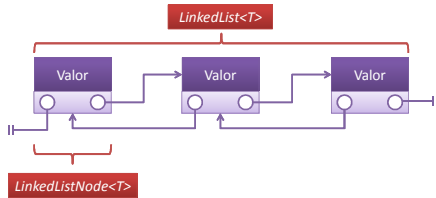
```
lista.Sort();
```

```
lista.Sort(new ListaComparer());
```

A Classe `LinkedList<T>`



- Representa uma lista duplamente encadeada



- Os elementos formam uma sequência, mas não possuem índices

Interagindo com Listas Encadeadas



- O inserção de elementos é feita com os métodos
 - `AddFirst()`
 - `AddLast()`
 - `AddBefore()`
 - `AddAfter()`

```
Linguagem linguagem = new Linguagem() { Nome = "C#" };  
LinkedListNode<Linguagem> node = lista.AddLast(linguagem);
```

Retorna o nó adicionado

Um `LinkedListNode<T>` possui as properties `Value`, `Previous` e `Next`

Interagindo com Listas Encadeadas



- A remoção de elementos não utiliza índices
 - `Remove()`
 - `RemoveFirst()`
 - `RemoveLast()`
- O acesso a elementos é feito pelas properties **First** e **Last** e pela iteração

```
foreach (Linguagem l in lista)  
{  
    //...  
}
```

List<T> X LinkedList<T>

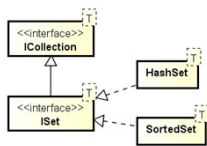


- A inserção de elementos no início ou meio de uma *LinkedList<T>* é mais rápido
 - Na *List<T>* os elementos do array devem ser movidos
 - Pode ser que todos os elementos precisem ser copiados para um novo array
- *List<T>* é mais rápida para buscar um elemento com base em um índice

Conjuntos



- Coleção onde os elementos são únicos



A Classe HashSet<T>



- Representa um conjunto de elementos únicos sem ordem definida

```
HashSet<string> conjunto = new HashSet<string>();
bool b = conjunto.Add("C#");
```

Retorna um booleano indicando se o elemento foi adicionado

Se um elemento já existe, ele não pode ser adicionado novamente

```
bool b = conjunto.Remove("C#");
```

Remove o elemento e retorna um booleano indicando se o elemento foi removido

Funcionamento do *HashSet<T>*

• Usa um algoritmo de hashing

Métodos *GetHashCode()* e *Equals()*

```
public override int GetHashCode()
{
    //...
}
```

O algoritmo de hashing deve retornar valores bem distribuídos

GetHashCode() → Encontra o índice
Equals() → Encontra o elemento dentro do índice

Métodos *GetHashCode()* e *Equals()*

- Se *GetHashCode()* for implementado, *Equals()* também deve ser
 - O ideal é implementar também a interface *IComparable<T>*
- Se dois objetos são iguais, os métodos *GetHashCode()* de ambos devem retornar o mesmo valor

A Classe *SortedSet*<T>



- Representa um conjunto ordenado de elementos únicos
 - A classe dos elementos deve implementar *IComparable*<T>
 - Um *IComparer*<T> pode ser utilizado

```
SortedSet<string> conjunto = new SortedSet<string>();  
conjunto.Add("VB");  
conjunto.Add("C#");
```

A ordenação é feita no momento da inserção

Métodos da Interface *ISet*<T>



- *HashSet*<T> e *SortedSet*<T> implementam a interface *ISet*<T>
- *ISet*<T> tem alguns métodos interessantes para trabalhar com conjuntos

Método	Descrição
a.Overlaps(b)	Verifica se existe interseção entre <i>a</i> e <i>b</i>
a.IsSubsetOf(b)	Verifica se <i>a</i> é um subconjunto de <i>b</i>
a.IsSupersetOf(b)	Verifica se <i>a</i> é um superconjunto de <i>b</i>
a.UnionWith(b)	Faz a união dos elementos de <i>a</i> e <i>b</i> e armazena o resultado em <i>a</i>
a.ExceptWith(b)	Remove de <i>a</i> todos os elementos que pertencem a <i>b</i>

HashSet<T> X *SortedSet*<T>

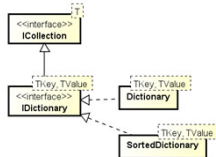


- Em termos de performance, tanto o *HashSet*<T> como o *SortedSet*<T> se comportam da mesma forma
- Prefira o *HashSet*<T>, a não ser que os elementos devam ser ordenados

Dicionários



- Coleção onde cada elemento é associado a uma chave
 - O elemento pode ser acessado através da sua chave
 - A chave não pode ser duplicada
- São conhecidos também como
 - Mapas (maps)
 - Tabelas hash (hash tables)



A Classe *Dictionary<TKey, TValue>*



- Representa um dicionário onde as chaves não são ordenadas

```
class Funcionario
{
    public string Nome { get; set; }
}
```

Tipo da chave Tipo do valor

```
Dictionary<string, Funcionario> dicionario =
    new Dictionary<string, Funcionario>();

Funcionario f1 = new Funcionario() { Nome = "Pedro" };
Funcionario f2 = new Funcionario() { Nome = "Joana" };
```

Inserindo e Removendo Elementos



- Existem duas formas para inserir elementos

```
dicionario.Add("AB3982", f1);
dicionario.Add("CD3709", f2);
```

Método Add()

```
dicionario["AB3982"] = f1;
dicionario["CD3709"] = f2;
```

Indexer

- A remoção é feita a partir da chave

```
bool b = dicionario.Remove("AB2982");
```

Remove() retorna um booleano, que indica se o elemento foi removido

Acessando Elementos



- Um elemento é acessado a partir da sua chave

```
Funcionario f = dicionario["AB3982"];
```

Indexer

KeyNotFoundException é lançada se a chave não existir

```
Funcionario f;  
bool b = dicionario.TryGetValue("AB3982", out f);
```

O booleano indica se a chave foi encontrada

Método `TryGetValue()`

Iterando Sobre os Elementos



- Obtendo as chaves

```
Dictionary<string, Funcionario>.KeyCollection keys = dicionario.Keys;  
foreach (string key in keys)  
{  
    //...  
}
```

- Obtendo os valores

```
Dictionary<string, Funcionario>.ValueCollection values = dicionario.Values;  
foreach (Funcionario value in values)  
{  
    //...  
}
```

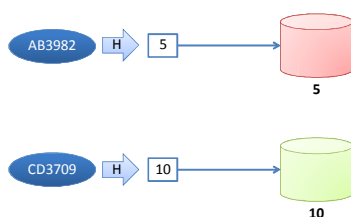
- Obtendo os pares de chave e valor

```
foreach (KeyValuePair<String, Funcionario> entry in dicionario)  
{  
    String key = entry.Key;  
    Funcionario func = entry.Value;  
    //...  
}
```

Armazenamento das Chaves



- As chaves são convertidas para um código hash para acelerar buscas futuras
 - A classe da chave deve implementar os métodos `Equals()` e `GetHashCode()`



A Classe *SortedDictionary<TKey, TValue>*

- Funciona de forma semelhante ao *Dictionary<TKey, TValue>*
- A diferença é que as chaves são ordenadas
 - A classe das chaves deve implementar *IComparable<T>*
 - Um objeto *IComparer<T>* pode ser utilizado

```
SortedDictionary<string, Funcionario> dicionario = new  
SortedDictionary<string, Funcionario>();
```

Dictionary<T, V> X *SortedDictionary<T, V>*

- *Dictionary<TKey, TValue>* é mais rápido para acessar, remover e buscar elementos
- Prefira o uso de *Dictionary<TKey, TValue>*, a não ser que as chaves precisem estar ordenadas

Listas X Conjuntos X Dicionários

- Listas
 - Permitem elementos duplicados
 - O acesso usando índices (*List<T>*) é bastante rápido
 - Operações inserção, remoção e busca podem ser lentas se comparadas aos conjuntos
- Conjuntos
 - Elementos devem ser únicos
 - Operações de inserção, remoção e busca são feitas muito rapidamente
- Dicionários
 - Elementos associados a uma chave

Filas

- Em uma fila, os elementos são processados da forma FIFO (*first in, first out*)

Enqueue
(enfileirar)

Dequeue
(desenfileirar)

A Classe `Queue<T>`

- Representa uma fila

```
Queue<char> q = new Queue<char>();
```

- Enfileirar elementos

```
q.Enqueue('A');
```

- Desenfileirar elementos

```
char c = q.Dequeue();
```

- Outros métodos/properties

```
int count = q.Count;
```

```
char c = q.Peek();
```

Conta elementos

Retorna o primeiro da fila


Pilhas

- Em uma pilha, os elementos são processados da forma LIFO (*last in, first out*)

Push
(empilhar)

Pop
(desempilhar)

A Classe *Stack<T>*



- Representa uma pilha

```
Stack<char> q = new Stack<char>();
```
- Empilhar elementos

```
q.Push('A');
```
- Desempilhar elementos

```
char c = q.Pop();
```
- Outros métodos/properties

```
int count = q.Count;  
char c = q.Peek();
```

Conta elementos

Retorna o primeiro da pilha



Softblue
