Capítulo 2
Trabalhando com Coleções

Introdução

Desde a primeira versão do .NET Framework existem classes que possibilitam armazenarmos outros objetos e fornecem toda a funcionalidade para interagir com estes objetos. Essas coleções obviamente foram mantidas, mas a Microsoft criou um novo subset de coleções chamado *Generics* que possibilita ao desenvolvedor escrever um código muito mais limpo e robusto.

No decorrer deste artigo veremos as coleções disponíveis dentro do .NET Framework 2.0. O capítulo começa mostrando as coleções existentes desde a versão 1.0 do mesmo e, avançando um pouco mais, analisaremos as coleções genéricas disponíveis na versão 2.0 que é uma funcionalidade que trouxe uma grande flexibilidade e uma melhoria notável em termos de performance e produtividade na escrita do código. Não veremos apenas as coleções concretas, mas também vamos entender qual a arquitetura disponível e as *Interfaces* necessárias e disponíveis para a criação de coleções customizadas. Ambas seções onde são abordados cada tipo de coleção, será abordado as Interfaces disponíveis para efetuarmos a comparação entre objetos.

Depois de analisar as coleção primárias e as coleções genéricas, vamos abordar as coleções especializadas, que são criadas para uma finalidade muito específica e também classes que fornecem as funcionalidades básicas para a criação de coleções customizadas para uma determinado cenário.

O que são Coleções?

Coleções são classes que armazenam em seu interior os mais diversos tipos de objetos. Elas são muito mais fácil de manipular em relação a *Arrays*, justamente porque coleções fornecem uma infraestrutura completa para a manipulação de seus elementos. Para citar alguns de seus benefícios, podemos destacar: redimensionamento automático; método para inserir, remover e verificar se existe um determinado elemento; métodos que permitem o usuário interceptar a inserção ou a exclusão de um elemento.

As coleções disponibilizadas pelo .NET Framework 2.0 e as *Interfaces* que são base para grande parte delas estão atualmente disponibilizadas em dois *namespaces* diferentes: **System.Collections** e **System.Collections.Generics**. O primeiro já existe desde a versão 1.0 do .NET Framework, mas o segundo é uma novidade que foi incorporada na versão 2.0.

Dentro destes *namespaces*, encontramos várias classes e *Interfaces* para a criação dos mais diversos tipos de coleções. As coleções estão definidas em três categorias: **Coleções Ordenadas**, **Coleções Indexadas** e Coleções baseadas em **"chave-e-valor"**. Veremos abaixo a finalidade de cada uma delas:

- **Coleções Ordenadas:** São coleções que são distingüidas apenas pela ordem de inserção e assim controla a ordem em que os objetos podem ser recuperados da coleção. **System.Collections.Stack** e **System.Collections.Queue** são exemplos deste tipo de coleção.
- **Coleções Indexadas:** São coleções que são distingüidas pelo fato de que seus valores e/ou objetos podem ser recuperados/acessados através de um índice numérico (baseado em 0 (zero)). **System.Collections.ArrayList** é um exemplo deste tipo de coleção.
- **Coleções "Chave-e-Valor":** Como o próprio tipo diz, é uma coleção que contém uma chave associado a algum tipo. Seus índices são classificados no valor da chave e podem ser recuperados na ordem classificado pela enumeração. *System.Collections.HashTable* é um exemplo deste tipo de coleção.

Nota: Para todos os tipos citados como exemplo para a descrição dos tipos de coleções sempre haverá um correspondente dentro do namespace **System.Collections.Generics**, que trabalhará de forma "genérica".

Coleções Primárias

Definimos como coleções primárias todas as coleções disponíveis dentro do *namespace* **System.Collections**. Estas coleções também são chamadas de *coleções fracamente tipadas* ou *coleções não genéricas*. Essas coleções, apesar de suportar grande partes das funcionalidades que são disponibilizadas pelo .NET Framework, há um grande problema, já que essas classes operam com tipos **System.Object**, o que causa problemas em termos de performance, pois se quiser fazer uma coleção de inteiros teremos que pagar pelo *boxing* e *unboxing*. Além disso, ainda há a questão de não trabalhar efetivamente com um único tipo, ou seja, o fato de possibilitar incluir um **System.Object**, está permitindo adicionar qualquer tipo de objeto dentro da coleção, o que pode causar problemas durante a execução da aplicação.

Para enterdemos a estrutura das coleções primárias, vamos analisar todas as *Interfaces* que foram implementadas nas coleções concretas e que também estão disponíveis para que o desenvolvedor possa customizar seus próprias coleções. A Imagem 2.1 exibe a hierarquia das *Interfaces* e a tabela a seguir detalha cada uma dessas *Interfaces*.

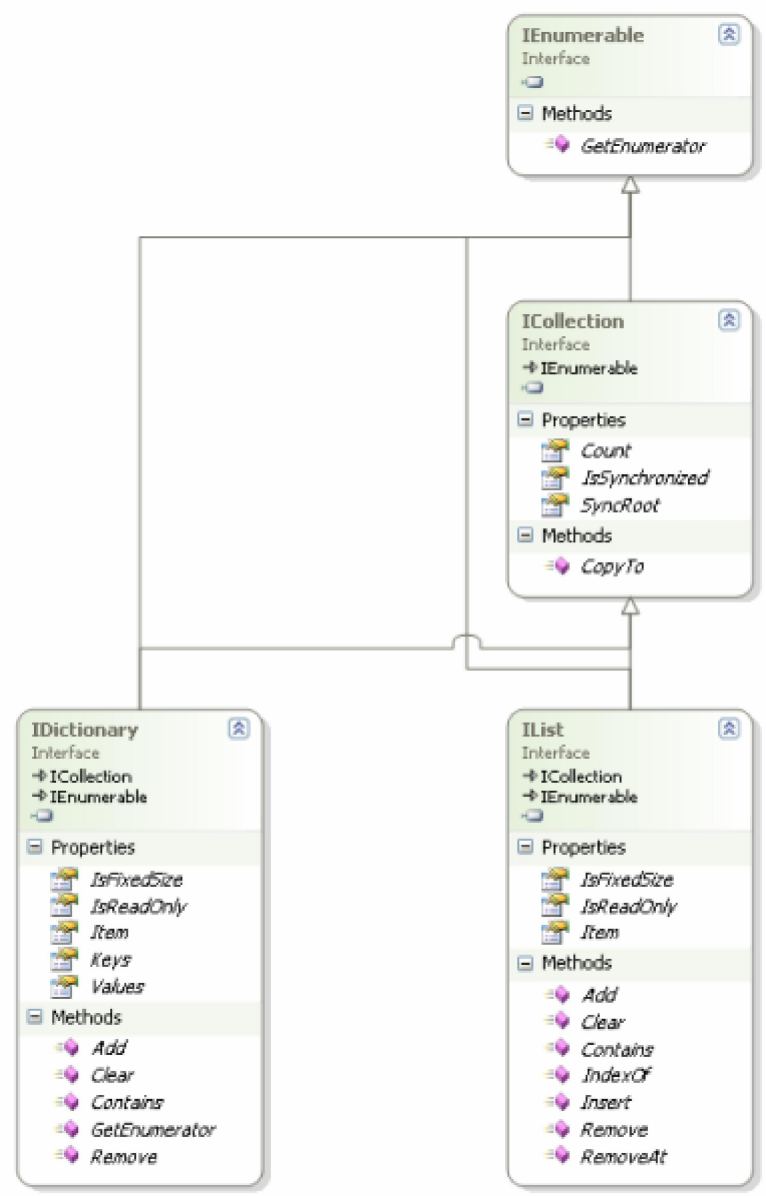


Imagem 2.1 – Hierarquia das *Interfaces* do *namespace System.Collections*

Interface	**Descrição**
ICollection	É a *Interface* base para todas as coleções que estão contidas dentro do *namespace* **System.Collections**. Direta ou indiretamente essas classes a implementa. Ela por sua vez é composta por três propriedades (*Count*, *IsSynchronized* e *SyncRoot*) e um método (*CopyTo*). *Interfaces* como **IList** e **IDictionary** derivam desta *Interface* **ICollection**, pois estas Interfaces são mais especializadas, tendo também outros métodos a serem definidos além dos quais já

	estão contidos dentro da *Interface* **ICollection**. Já as classes como **System.Collections.Queue** e **System.Collections.Stack** implementam a *Interface* **ICollection** diretamente.
IComparer	Esta *Interface* é utilizada para customizar uma ordenação (*sorting*) em uma determinada coleção. A *Interface* **IComparer** é composta por um método chamado *Compare*, qual compara dois objetos e retorna um valor inteiro indicando se os objetos são ou não iguais, ou seja, se os valores comparados forem iguais, 0 (zero) é retornado. Há classes, como por exemplo o **ArrayList**, que já implementa o método *Sort* para tal ordenação, mas há casos em que desejamos criar uma ordenação customizada, onde a ordenação padrão não nos interessa. É justamente neste momento que a *Interface* **IComparer** é utilizada, inclusive o método *Sort* da classe **ArrayList** tem um *overload* que recebe como parâmetro um objeto do tipo *Interface* **IComparer**. Quando utilizamos o método *Sort* do **ArrayList** sem informar nenhum parâmetro, a ordenação é feita em ordem ascendente, a padrão. Mas podemos querer ordenar em ordem descendente, e com isso teríamos que criar uma classe que implementa a *Interface* **IComparer**, codificando assim o método *Compare*.
IDictionary	A *Interface* **IDictionary** é a base para todas as coleções do tipo "chave-e-valor". Cada elemento inserido neste tipo de coleção é armazenado em um objeto do tipo **DictionaryEntry** (*Structure*). Cada associação deve ter um chave original que não seja uma referência nula, mas o seu respectivo valor de associação pode incluir sem problemas uma referência nula.
IDictionaryEnumerator	Esta *Interface* tem a mesma finalidade da *Interface* **IEnumerator**, ou seja, percorrer e ler os elementos de uma determinada coleção, mas esta em especial, trata de enumerar os elementos de um dicionário, ou seja, uma coleção que contenha "chave-e-valor".
IEnumerable	É a base direta ou indiretamente para algumas outras *Interfaces* e todas as coleções a implementam. Esta interface tem apenas um método a ser implementado, chamado *GetEnumerator*, qual retorna um objeto do tipo da *Interface* de **IEnumerator**.
IEnumerator	É a base para todos os enumeradores. Composta por uma propriedade (*Current*) e dois métodos (*MoveNext* e *Reset*), percorre e lê todos os elementos de uma determinada coleção, permitindo apenas ler os dados, não podendo alterá-los. Vale levar em consideração que enumeradores não podem ser

	utilizados para a coleção subjacente.
IEqualityComparer	Expõe os métodos chamados *Equals* e *GetHashCode* que permitem a comparação entre dois objetos quando a igualdade. *Esta Interface foi introduzida a partir do .NET Framework 2.0.*
IList	Derivada das *Interfaces* **ICollection** e **IEnumerable** ela é a *Interface* base para todas as listas contidas no .NET Framework. Listas quais podemos acessar individualmente por um determinado índice. As implementações da *Interface* **IList** podem estar em uma das seguintes categorias: **Read-Only**, **Fixed-Size** e **Variable-Size**. Uma **IList** *Read-Only*, não permite que você modifique os elementos. Já uma **IList** *Fixed-Size*, não permite adicionar ou remover os elementos, mas permite que você altere os elementos existentes. E por fim, uma **IList** *Variable-Size*, qual permite que você adicione, remova ou altere os elementos. Claro que quando implementamos esta *Interface* em alguma classe, é necessário criarmos um membro privado que será o responsável por armazenar os items. Este membro privado é manipulado pelos métodos e propriedades da *Interface* **IList** que serão implementados nesta classe.

Antes de efetivamente entrar em cada uma das coleções concretas dentro das coleções primárias, vamos analisar a implementação de alguma das *Interfaces* que vimos acima para um cenário customizado de uma determinada aplicação.

Entre as várias Interfaces acima mostradas, duas delas praticamente trabalham em conjunto: **IEnumerable** e **IEnumerator**. A Interface **IEnumerator** fornece métodos para iterar pela coleção; já a *Interface* **IEnumerable** deve ser implementada em um tipo que você deseja iterar pela coleção através de um loop **foreach** (**For Each** em Visual Basic .NET). A implementação destas Interfaces é geralmente feito em classes separadas e, na maioria dos casos, o método *GetEnumerator* da *Interface* **IEnumerable** retorna a instância da classe privada que implementa **IEnumerator**. O código abaixo exemplifica como iterar pelas cidades categorias contidas dentro da classe *CategoriesEnumerator*:

```
VB.NET
Public Class CategoriesEnumerator

    Implements IEnumerator

    Private _categories() As String = _
        {"ASP.NET", "VB.NET", "C#", "SQL", "XML"}
    Private _current As Integer = -1
```

```
    Public ReadOnly Property Current() As Object Implements
IEnumerator.Current
        Get
            If Me._current < 0 OrElse Me._current > 4 Then
                Throw New InvalidOperationException
            Else
                Return Me._categories(Me._current)
            End If
        End Get
    End Property

    Public Function MoveNext() As Boolean Implements
IEnumerator.MoveNext
        Me._current += 1
        Return Me._current <= 4
    End Function

    Public Sub Reset() Implements IEnumerator.Reset
        Me._current = -1
    End Sub

End Class
```

C#

```
public class CategoriesEnumerator : IEnumerator
{
    private string[] _categories =
        new string[] { "ASP.NET", "VB.NET", "C#", "SQL", "XML" };
    private int _current = -1;

    public object Current
    {
        get
        {
            if (this._current < 0 || this._current > 4)
                throw new InvalidOperationException();
            else
                return this._categories[this._current];
        }
    }

    public bool MoveNext()
    {
        this._current++;
        return this._current <= 4;
    }

    public void Reset()
    {
```

```
        this._current = -1;
    }
}
```

A classe acima está finalizada. Se quiséssemos exibir os valores na tela, bastaria loop verificando se o método retorna *True* ou *False* e exibir o conteúdo da tela recuperand-o através da propriedade *Current*. Mas e se quiséssemos iterar por essa coleção através de um laço *For Each*? É nesta ocasião que a *Interface* **IEnumerable** entra em ação. É necessário que você cria uma classe auxiliar, implementando a *Interface* em questão onde, dentro do método *GetEnumerator* fornecido por ela, retornará uma instância da classe *CategoriesEnumerator*. O código abaixo mostra essa implementação na íntegra e a respectiva utilização:

VB.NET

```
Public Class Categories
    Implements IEnumerable

    Public Function GetEnumerator() As IEnumerator Implements
IEnumerable.GetEnumerator
        Return New CategoriesEnumerator()
    End Sub
End Class

'Utilização:

For Each category As String In New Categories()
    Console.WriteLine(category)
Next
```

C#

```
public class Categories : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        return new CategoriesEnumerator();
    }
}

//Utilização:

foreach (string category in new Categories())
    Console.WriteLine(category);
```

Outra *Interface* que é comumente utilizada é a **IList**. Ela fornece grande parte da infraestrutura para todas as coleções do .NET Framework. Um exemplo de implementação para esta Interface é exibido abaixo:

VB.NET

```
Public Class TestIList

    Implements IList

    Private _arr As ArrayList

    Public Sub New()
        Me._arr = New ArrayList
    End Sub

    Public ReadOnly Property Count() As Integer Implements
System.Collections.ICollection.Count
        Get
            Return Me._arr.Count()
        End Get
    End Property

    Public Function Add(ByVal value As Object) As Integer
Implements System.Collections.IList.Add
        Return Me._arr.Add(value)
    End Function

    ' outros métodos ocultados

End Class
```

C#

```
public class TestIList : IList
{
    private ArrayList _arr;

    public TestIList()
    {
        this._arr = new ArrayList();
    }

    public int Count
    {
        get
        {
            return this._arr.Count;
        }
    }
    public int Add(object value)
```

```
    {
        return this._arr.Add(value);
    }
}
```

É importante ressaltar que no código acima é criado um membro privado chamado de ***_arr*** do tipo **ArrayList** e que sua manipulação é feita através dos métodos que estão definidos na *Interface* **IList** e nesta classe implementados.

Com a *Interface* **IList** conseguimos criar uma coleção customizada, podendo inclusive criar uma coleção fortemente tipada, onde seus tipos seriam únicos, ou seja, seria aceito apenas um objeto de um tipo definido e assim, em design-time já conseguiríamos detectar possíveis problemas de *casting*, mas infelizmente não evita o *boxing* e *unboxing*.

O código acima é apenas um exemplo para entedermos o funcionamento da implementação da *Interface* **IList**, onde a implementamos e criamos um membro privado do tipo **ArrayList** para armazenar os objetos, pois dentro do *namespace* **System.Collections** já temos um classe abstrata, chamada **CollectionBase**, que é a base para a criação de coleções fortemente tipadas. Veremos mais tarde, ainda neste capítulo, sobre a coleção **CollectionBase**.

Coleções

ArrayList

A coleção **ArrayList** é um dos objetos mais tradicionais que o .NET Framework expõe. Trata-se de uma classe que implementa a *Interface* **IList** e é muito similar a um *Array* unidemensional, mas automaticamente redimesiona o seu tamanho (que inicialmente é 0) de acordo com a necessidade.

Através do método *Add*, podemos adicionar qualquer tipo de objeto, sendo uma referência nula, um objeto já adicionado anteriormente ou de tipos diferentes. Depois de adicionados, seus items podem ser acessados através de um índice inteiro (propriedade (***indexer***) fornecido pela *Interface* **IList**), que indica a posição do elemento que quer recuperar. Trata-se de uma coleção baseada em 0 (zero) e, sendo assim, a posição do primeiro elemento é 0.

Para exibir um exemplo do uso da classe ArrayList é um tanto quanto simples:

```
VB.NET
Dim categorias As New ArrayList()
categorias.Add("ASP.NET")
categorias.Add("VB.NET")
categorias.Add("C#")
```

```
categorias.Add("XML")

'Utilização:

For index As Integer = 0 To categorias.Count - 1
    Console.WriteLine(categorias(index))
Next


C#
ArrayList categorias = new ArrayList();
categorias.Add("ASP.NET");
categorias.Add("VB.NET");
categorias.Add("C#");
categorias.Add("XML");

for (int index = 0; index < categorias.Count; index++)
    Console.WriteLine(categorias[index]);
```

Stack

A coleção Stack é uma coleção do tipo **LIFO** (*last-in-first-out*), ou seja, o primeiro item a entrar na coleção é o último a sair. Ela oferece dois métodos importantes: *Push* e *Pop*. O primeiro encarrega-se de adicionar o item a coleção. Já o segundo, *Pop*, é utilizado para recuperar e remover o último item adicionado na coleção. Além destes dois métodos, ainda existe o método *Peek*, que trabalha basicamente da mesma forma que o método *Pop*, ou seja, recupera o último item adicionado, mas com uma grande diferença: não remove o item da coleção.

Para exibir um exemplo da utilização da coleção *Stack*, analise o código abaixo:

```
VB.NET
Dim stack As New Stack()
stack.Push("1")
stack.Push("2")
stack.Push("3")
stack.Push("4")

Console.WriteLine("Quantidade: " & stack.Count)
While(stack.Count > 0)
    Console.WriteLine(stack.Pop());
End While
Console.WriteLine("Quantidade: " & stack.Count)

'Output:
Quantidade: 4
4
```

```
3
2
1
Quantidade: 0

C#
Stack stack = new Stack();
stack.Push("1");
stack.Push("2");
stack.Push("3");
stack.Push("4");

Console.WriteLine("Quantidade: " + stack.Count);
while (stack.Count > 0)
    Console.WriteLine(stack.Pop());

Console.WriteLine("Quantidade: " + stack.Count);

//Output:
Quantidade: 4
4
3
2
1
Quantidade: 0
```

Queue

A coleção Queue trata-se de uma coleção do tipo **FIFO** (*first-in-first-out*), ou seja, o primeiro a entrar na lista é o primeiro a sair. Há alguns programas que trabalham neste formato, como por exemplo o *Message Queue* do Windows. Esse tipo de coleções são úteis quando necessitamos trabalhar em um ambiente que exige um processamento sequencial.

Podemos fazer uma analogia a qualquer tipo de fila de qualquer estabelecimento. Exemplo: em um banco, onde somente exista um caixa atendendo uma fila de N clientes, o primeiro a chegar, que obviamente está no início da file, será o primeiro a ser atendido e, conseqüentemente, o primeiro a sair e, o processo continua até o término da fila.

Assim como na coleção **Stack**, a **Queue** também possui dois métodos importantes: *Enqueue* e *Dequeue*. O primeiro encarrega-se de enfileirar o valor a coleção. Já o segundo, *Dequeue*, é utilizado para recuperar e remover o item mais antigo adicionado na coleção. Novamente temos o método *Peek*, que trabalha basicamente da mesma forma que o método *Dequeue*, ou seja, recupera o item mais antigo adicionado, mas com uma grande diferença: não remove o item da coleção.

Para exibir um exemplo da utilização da coleção *Queue*, analise o código abaixo:

```
VB.NET
Dim queue As New Queue()
queue.Enqueue("1")
queue.Enqueue("2")
queue.Enqueue("3")
queue.Enqueue("4")

Console.WriteLine("Quantidade: " & queue.Count)
While(queue.Count > 0)
    Console.WriteLine(queue.Dequeue());
End While
Console.WriteLine("Quantidade: " & queue.Count)

'Output:
Quantidade: 4
4
3
2
1
Quantidade: 0

C#
Queue queue = new Queue();
queue.Enqueue("1");
queue.Enqueue("2");
queue.Enqueue("3");
queue.Enqueue("4");

Console.WriteLine("Quantidade: " + queue.Count);
while (queue.Count > 0)
    Console.WriteLine(queue.Dequeue());

Console.WriteLine("Quantidade: " + queue.Count);

//Output:
Quantidade: 4
4
3
2
1
Quantidade: 0
```

Hashtable

Esta classe representa uma coleção do tipo chave/valor e são organizadas de acordo com um valor *hash* da chave informada e armazenada em uma estrutura do tipo

DictionaryEntry. Como é calculado um valor *hash* em cima da chave informada, a performance é beneficiada, pois reduz o número de comparações efetuadas quando se procura por uma determinada chave, pois o *Hashtable* extrai o código *hash* do valor a ser encontrado e submete este valor para a pesquisa. Certifique-se sempre antes de adicionar um item a coleção de que esta chave já não existe, pois do contrário, uma exceção do tipo **ArgumentException** é atirada.

Como o **Hashtable** armazena o código *hash* para a chave informada, não altere este valor, senão o **Hashtable** não será mais capaz de encontrá-lo. Se a sua chave tende a mudar, então o correto é remover o item da coleção, alterar a chave e, em seguida, adicioná-lo novamente. A chave nem sempre precisa ser uma *string*. Qualquer objeto que implemente o método **System.Object.GetHashCode** e **System.Object.Equals** será permitido. É importante saber que a chave não pode ser uma referência nula, ao contrário do valor, que permite isso.

Para exibir um exemplo simples da utilização da coleção **Hashtable**, analise o código abaixo:

```
VB.NET
Dim table As new Hashtable()
table.Add("AC", "Acre")
table.Add("RJ", "Rio de Janeiro")
table.Add("SP", "São Paulo")

For Each entry As DictionaryEntry In table
    Console.WriteLine(entry.Key & " - " & entry.Value)
Next

C#
Hashtable table = new Hashtable();
table.Add("AC", "Acre");
table.Add("RJ", "Rio de Janeiro");
table.Add("SP", "São Paulo");

foreach (DictionaryEntry entry in table)
    Console.WriteLine(entry.Key + " - " + entry.Value);
```

Como mencionamos anteriormente, os objetos adicionados dentro de uma coleção **Hashtable** são do tipo **DictionaryEntry**, o que significa que para iterar entre os valores do mesmo, terá que gerar o laço como exemplicado no código acima.

SortedList

A coleção **SortedList** é basicamente uma coleção **Hashtable**, com uma diferença: é uma coleção ordenada pela chave. Evidentemente que a **SortedList** tende ser mais lenta que a **Hashtable**, devido ao *overhead* de *sorting*.

Quando um item é adicionado em uma **SortedList**, ele já é inserido exatamente no local correto e os índices são ajustados automaticamente. O mesmo acontece quando o elemento é removido da coleção. Entretanto, é importante observar que o índice de um determinado item pode variar, pois a coleção é reordenada e, se tiver um índice para um objeto específico, ele já pode não mais apontar para o mesmo.

Para exibir um exemplo simples da utilização da coleção **SortedList**, analise o código abaixo:

```
VB.NET
Dim list As new SortedList()
list.Add("RJ", "Rio de Janeiro")
list.Add("AC", "Acre")
list.Add("SP", "São Paulo")

For Each entry As DictionaryEntry In list
    Console.WriteLine(entry.Key & " - " & entry.Value)
Next

'Output
AC - Acre
RJ - Rio de Janeiro
SP - São Paulo

C#
SortedList list = new SortedList();
list.Add("RJ", "Rio de Janeiro");
list.Add("AC", "Acre");
list.Add("SP", "São Paulo");

foreach (DictionaryEntry entry in list)
    Console.WriteLine(entry.Key + " - " + entry.Value);

//Output
AC - Acre
RJ - Rio de Janeiro
SP - São Paulo
```

Além dos items da coleção *SortedList* serem acessíveis através da chave, é também possível acessá-lo através do índice. Para isso, utilize o método *GetByIndex* passando como parâmetro um número inteiro que corresponde ao elemento que quer recuperar.

CollectionBase

É muito comum em runtime retornarmos valores da Base de Dados e armazenarmos em objetos do tipo *Datasets*, mas com isso ocorre o "*Weakly Typed*" e o "*Late Binding*", ou seja, não temos a segurança de tipos e o acesso à suas propriedades somente ocorre durante o tempo de execução.

Tendo esse problema, o que podemos fazer para termos nossos próprios objetos com suas respectivas propriedades e métodos ao invés de uma genérica? Eis o momento que entra em cena a classe **CollectionBase**.

A classe **CollectionBase** é uma classe onde podemos apenas herdá-la, ou seja, é uma classe abstrata. Ela implementa as três seguintes *Interfaces*: **IList**, **ICollection** e **IEnumerable** que definem os métodos que permitem aceitar tipos **System.Object**. Além disso, ela ainda encapsula um objeto do tipo **ArrayList**, onde ficarão armazenados os elementos da coleção. Entre os principais métodos desta classe, temos:

Método	Descrição
Add	Adiciona um novo elemento na coleção e retorna o indíce (posição) que o objeto foi adicionado.
Contains	Verifica se já existe um determinado elemento dentro da coleção e retorna um valor boleano indicando ou não sua existência.
Insert	Insere um elemento na coleção em uma determinada posição.
Item	Retorna ou recebe um elemento dado uma posição.
Remove	Remove um elemento da coleção.
RemoveAt	Remove um objeto da coleção dado uma posição.

Os métodos acima são expostos através da propriedade *List*. Será necessário criar esses métodos que servirão de *wrapper* para os métodos internos de manipulação da coleção. A direferença é que esses métodos que serão criados e expostos para o cliente devem aceitar em seus parâmetros o mesmo tipo, isso quer dizer que, se a coleção trabalhará somente com objetos do tipo *Usuario*, é necessário que os membros *Add*, *Remove*, *Contains* e *Item* recebam este mesmo tipo para garantir o *type-safe*. O trecho de código abaixo mostra como a classe deve ser implementada:

```
VB.NET
Public Class UsuarioColecao
    Inherits CollectionBase

    Public Function Add(ByVal u As Usuario) As Integer
        Return MyBase.List.Add(u)
    End Function

    Public Function Contains(ByVal u As Usuario) As Boolean
        Return MyBase.List.Contains(u)
```

```
    End Function

    Public Sub Insert(ByVal index As Integer, _
        ByVal u As Usuario)

        MyBase.List.Insert(index, u)
    End Sub

    Default Public Property Item(ByVal index As Integer) As
Usuario
        Get
            Return CType(MyBase.List(index), Usuario)
        End Get
        Set(ByVal Value As Usuario)
            MyBase.List(index) = Value
        End Get
    End Property

    Public Sub Remove(ByVal u As Usuario)
        MyBase.List.Remove(u)
    End Sub
End Class
```

C#

```
public class UsuarioColecao : CollectionBase
{
    public int Add(Usuario u)
    {
        return base.List.Add(u);
    }

    public bool Contains(Usuario u)
    {
        return base.List.Contains(u);
    }

    public void Insert(int index, Usuario u)
    {
        base.List.Insert(index, u);
    }

    public Usuario this[int index]
    {
        get
        {
            return (Usuario)base.List[index];
        }
        set
        {
            base.List[index] = value;
```

```
        }
    }

    public void Remove(Usuario u)
    {
        base.List.Remove(u);
    }
}
```

Durante a sua utilização, ela somente poderá manipular objetos do tipo *Usuario*. Qualquer outro tipo que você tentar adicionar resultará em um erro de compilação e não conseguirá compilar o projeto até que o problema seja sanado. Isso irá resolver o problema do *type-safe*, já que ao invés de expor objetos do tipo **ArrayList**, você pode optar por expor a coleção customizada.

Apesar de uma boa melhora, ainda temos alguns problemas que nos causam prejuízo: a produtividade é muito baixa, já que, se tivéssemos 20 coleções de objetos diferentes, devemos ter 20 coleções distintas. Além disso, o problema de *boxing/unboxing* continua existindo, já que internamente a classe **CollectionBase** armazena em um objeto **ArrayList**. Esses problemas foram todos abolidos com a introdução dos *Generics*, que veremos ainda neste capítulo.

DictionaryBase

Assim como a classe **CollectionBase**, a **DictionaryBase** vem para forneceer uma infraestrutura base para a criação de dicionários (pares de chave-valor) com tipos específicos que você desejar, para novamente garantir o *type-safe*.

Internamente esta coleção armazena os itens em uma outra coleções do tipo **Hashtable**, que já vimos acima. A implementação é basicamente a mesma em relação ao que vimos acima com a coleção **CollectionBase**, apenas mudando que temos uma coleção do tipo **chave-valor** que, por sua vez, armazenará uma *string* como *chave* e um objeto do tipo *Usuario* como *valor*:

```
VB.NET
Public Class UsuarioDictionary
    Inherits DictionaryBase

    Public Sub Add(ByVal key As String, ByVal value As Usuario)
        MyBase.Dictionary.Add(key, value)
    End Function

    'Outros métodos
End Class
```

```
C#
public class UsuarioDictionary : DictionaryBase
{
    public void Add(string key, Usuario value)
    {
        base.Dictionary.Add(key, value);
    }

    //Outros métodos
}
```

Com relação a forma de iteração, continua sendo a mesma da **Hashtable**, ou seha, através da estrutura **DictionaryEntry**. Mais uma vez, isso somente ajudará no que diz respeito ao *type-safe*. A produtividade e performance ainda continuam comprometidas.

Coleções Genéricas

Na seção anterior vimos a infraestrutura e a utilização das mais diversas coleções disponíveis no .NET Framework desde a sua versão 1.0. Como também foi dito, temos alguns problemas quando fazemos das coleções primárias, pois havia sempre *boxing* e *unboxing* para adicionar, recuperar e remover os elementos de cada uma das coleções. Um outro agravante é que para termos uma coleção "fortemente tipada" tínhamos que escrever uma porção de código para garantir o "*type-safe*", mas mesmo assim, não evitava o *boxing/unboxing*, prejudicando imensamente a performance.

Felizmente a Microsoft introduziu as coleções genéricas. Essas coleções são basicamente as mesmas que vimos anteriormente com uma enorme diferença: operam com qualquer tipo definido dentro do .NET Framework ou um tipo customizado pelo desenvolvedor. Além disso, não há mais *boxing/unboxing* e também não exige mais o *casting* quando necessitar recuperar um dos elementos da coleção, já que toda a checagem de tipos é efetuada durante a escrita do código, ou seja, se criar uma coleção genérica especificando o tipo *string* para este coleção, jamais conseguirá adicionar um tipo de dado inteiro ou decimal.

Interfaces disponíveis

Foi introduzido um novo *namespace* chamado de **System.Collections.Generic**. Dentro deste namespace há todas as *Interfaces* primárias, só que em sua forma genérica. Isso quer dizer que temos todas essas *Interfaces* agoram operando com um tipo genérica. Basicamente dentro de cada *Interface* genérica temos o(s) mesmo(s) membro(s) que possuem as *Interfaces* primárias, só que estes tipos são substituídos pelo tipo que o desenvolvedor desejar manipular.

Como a maioria das *Interfaces* tem exatamente a mesma finalidade das *Interfaces* primárias, apenas operando com um tipo específico, as mesmas serão apenas citadas abaixo sem suas definições, mas sempre com suas correspondentes primárias:

Interface Genérica	**Correspondente Primária**
ICollection<T>	ICollection
IComparer<T>	IComparer
IDictionary<TKey, TValue>	IDictionary
IEnumerable<T>	IEnumerable
IEnumerator<T>	IEnumerator
IEqualityComparer<T>	IEqualityComparer
IList<T>	IList

Cada uma destas *Interfaces* são implementadas nas mais diversas coleções genéricas, quais serão abordadas ainda neste capítulo. É importante dizer que estas *Interfaces*, com as *Interfaces* primárias, estão a disposição para que o desenvolvedor possa criar classes (coleções) mais customizadas, de acordo com sua necessidade. Na tabela acima, o "T" deverá ser substituído por um tipo que você desejar que essa *Interface* manipule. Uma das únicas exceções é que a *Interface* **IDictionary** necessita de dois tipos: um para a chave e outro para o valor. Isso irá possibilitar o utilizador da *Interface* determinar qual será o tipo de dado da chave e o tipo de dado do valor que deverá ser armazenado pela coleção.

Para implementar uma dessas *Interfaces*, devemos utilizar a seguinte sintaxe:

```
VB.NET
Imports System.Collections.Generic

Public Class ListaUsuarios
    Implements IList(Of Usuario)

    '...
End Class

C#
using System.Collections.Generic;

public class ListUsuarios : IList<Usuario>
{
    //...
}
```


Coleções

List<T>

Uma das mais populares coleções fornecidas pelo *namespace* das coleções genéricas, **System.Collections.Generic**, é a coleção **List<T>**. Essa coleção pode ser comparada ao **ArrayList**, devido as suas finalidades, só que a coleção **List<T>** permite que você especifique um tipo, que a coleção toda irá manipular, o que garantirará que todas as propriedades e métodos desta classe somente aceitarão objetos do tipo especificado na construção da coleção, ao contrário do **ArrayList** que, por sua vez, aceita um **System.Object**, em outras palavras, "qualquer coisa", gerando todos os problemas que já discutimos acima.

Assim como o **ArrayList**, a coleção **List<T>** permite acessar seu itens através de um índice, métodos para efetuar buscas, ordenação e a manipulação completa a coleção e seus respectivos itens. Ambas as coleções implementam as *Interfaces* **IList** e **IList<T>**, respectivamente. Entre os vários métodos e propriedades fornecidos pela coleção **List<T>** podemos citar os mais importantes que, operam sempre com o tipo especificado durante a criação da coleção:

Método	Descrição
Contains	Dado um objeto, ele retorna um valor booleano indicando se o mesmo existe ou não dentro da coleção corrente.
IndexOf	Dado um objeto, ele faz a busca dentro da coleção corrente e, se o encontrar, retorna um número inteiro indicando a posição do elemento. Se não for encontrado, -1 é retornado.
Sort	Como o próprio nome diz, é utilizado para permitir a ordenação dos itens que estão contidos na coleção. Este método, em um dos seus *overloads*, aceita uma instância de um objeto que implementa a *Interface* **IComparer<T>** que poderá determinar os critérios de ordenação da coleção.
Count	Retorna um número inteiro indicando o número de elementos dentro da coleção. Se não existir nenhum elemento, 0 é retornado.
Item	Através da propriedade *default* (*indexer* em Visual C#), podemos recuperar ou atribuir um valor para um determinado item da coleção.
Clear	Quando invocado, remove todos os itens da coleção.

Através do código abaixo, podemos analisar como devemos proceder para a utilização da coleção **List<T>**:

```
VB.NET
Imports System.Collections.Generic

Dim lista As New List(Of String)
lista.Add("Generics é legal!")
lista.Add("Visual Basic .NET suporta Generics!")

lista.Add(123) 'gera um erro
```

```
C#
using System.Collections.Generic;

List<string> lista = new List<string>();
lista.Add("Generics é legal!");
lista.Add("Visual C# suporta Generics!");

lista.Add(123); // gera um erro
```

Nota Importante: Como a coleção **List<T>** é otimizada apenas para execução de algumas tarefas onde ter uma boa performance é crucial. Sendo assim, você nunca deve retornar uma coleção do tipo **List<T>** em suas API's de *object models* e sim uma coleção do tipo **Collection<T>**. A coleção do tipo **List<T>** não permite que você receba notificações quando o cliente modificar a coleção.

Collection<T>

A classe/coleção **Collection<T>** está localizada dentro do *namespace* **System.Collections.ObjectModel**. Ela fornece toda a base para a criação de uma coleção genérica e, apesar de não estar em um *namespace* "dentro" de *Generic*, ela é uma coleção genérica da mesma forma.

Para utilização dela, você pode criar diretamente a instância dela, definindo o tipo que ela irá operar ou, se desejar, ter um tipo específico que representará a coleção. Neste segundo caso, você pode herdar diretamente de **Collection<T>** e já especificar o tipo. Assim, você poderá expor essa coleção aos clientes e ainda, como já foi falado acima, a coleção **Collection<T>** é flexível, pois permite interceptarmos a inserção ou remoção de algum elemento da coleção, limpeza, definição de algum elemento mas, para isso, será mesmo necessário herdar em sua classe (coleção) e sobrescrever os métodos que permitem isso, como por exemplo o método *RemoveItem*.

Como existem duas formas de utilizarmos essa coleção, vamos analisar os dois tipos, sendo o primeiro a utilização direta da coleção **Collection<T>** e, em seguida, essa mesma classe será herdada em uma coleção mais customizada, onde poderemos interceptar algumas operações sob ela.

```
VB.NET
Imports System.Collections.ObjectModel

Dim coll As New Collection(Of String)
coll.Add("Generics é legal!")
coll.Add("Visual Basic .NET suporta Generics!")

C#
```

```
using System.Collections.ObjectModel;

Collection<string> coll = new Collection<string>();
coll.Add("Generics é legal!");
coll.Add("Visual C# suporta Generics!");
```

VB.NET

```
Imports System.Collections.ObjectModel

Module Module1
    Sub Main()
        Dim usuarios As New ColecaoUsuario()
        AddHandler usuarios.Removed, AddressOf Removido
        usuarios.Add("José")
        usuarios.Add("João")
        usuarios.Add("Augusto")
        usuarios.Add("Israel")
        usuarios.Add("Marcelo")
        usuarios.Add("Claudia")
        usuarios.Add("Leandro")
        usuarios.Remove("Israel")
    End Sub

    Private Sub Removido(ByVal sender As Object, ByVal e As
RemovedItemEventArgs)
        Console.WriteLine("Item removido: " + e.RemovedItem)
    End Sub
End Module

Public Class ColecaoUsuario
    Inherits Collection(Of String)

    Public Event Removed As EventHandler(Of RemovedItemEventArgs)

    Protected Overrides Sub RemoveItem(ByVal index As Integer)
        Dim removedItem As String = Me(index)
        MyBase.RemoveItem(index)

        Dim args As New RemovedItemEventArgs(removedItem)
        RaiseEvent Removed(Me, args)
    End Sub

End Class

Public Class RemovedItemEventArgs
    Inherits EventArgs

    Private _removedItem As String
```

```
    Public Sub New(ByVal removedItem As String)
        Me._removedItem = removedItem
    End Sub

    Public ReadOnly Property RemovedItem() As String
        Get
            Return Me._removedItem
        End Get
    End Property
End Class
```

C#

```
using System.Collections.ObjectModel;

class Program
{
    static void Main(string[] args)
    {
        ColecaoUsuario usuarios = new ColecaoUsuario();
        usuarios.Removed +=
            new EventHandler<RemovedItemEventArgs>(Removido);

        usuarios.Add("José");
        usuarios.Add("João");
        usuarios.Add("Augusto");
        usuarios.Add("Israel");
        usuarios.Add("Marcelo");
        usuarios.Add("Flavia");
        usuarios.Add("Leandro");
        usuarios.Remove("Israel");
    }

    static void Removido(object sender,
        Program.RemovedItemEventArgs e)
    {
        Console.WriteLine("Item removido: " + e.RemovedItem);
    }

    public class ColecaoUsuario : Collection<string>
    {
        public event EventHandler<RemovedItemEventArgs> Removed;

        protected override void RemoveItem(int index)
        {
            string removedItem = this.Items[index];
            base.RemoveItem(index);

            if (Removed != null)
            {
```

```
                RemovedItemEventArgs args =
                    new RemovedItemEventArgs(removedItem);
                this.Removed(this, args);
            }
        }
    }

    public class RemovedItemEventArgs : EventArgs
    {
        private string _removedItem;

        public RemovedItemEventArgs(string removedItem)
        {
            this._removedItem = removedItem;
        }

        public string RemovedItem
        {
            get
            {
                return this._removedItem;
            }
        }
    }
}
```

Como podemos analisar neste segundo caso, foi criado uma classe chamada *ColecaoUsuario* que herda diretamente da classe **Collection<T>**, especificando o tipo *string*. Com isso, temos acesso a todos os métodos fornecidos pela classe base. A idéia é que quando um elemento for removido da coleção, uma mensagem seja exibida ao usuário que o respectivo item foi removido. Para isso, iremos sobrescrever o método **RemoveItem**, que é denotado como *protected virtual* na classe base. Isso irá permitir interceptar o processo de remoção e, via eventos, podemos notificar o cliente que a coleção foi alterada, algo que não é possível fazer com a classe **List<T>**.

Stack<T> e Queue<T>

Ambas as coleções tem exatamente a mesma finalidade que as coleções primárias **Stack** e **Queue**. A única diferença entre elas é que as classes **Stack<T>** e **Queue<T>** operam da forma genérica. Isso quer dizer que, seus principais métodos: *Push* e *Pop*, *Enqueue* e *Dequeue*, respectivamente, inserem ou retornam sempre o tipo especificado durante a criação da coleção. Em relação as coleções primária, na delcaração precisamos especificar o tipo, asim como já é necessário com a classe **List<T>**.


```
VB.NET
Imports System.Collections.Generic
```

```
Dim coll As New Stack(Of String)

C#
using System.Collections.Generic;

Stack<string> coll = new Stack<string>();
```

Dictionary<TKey, TValue>

Essa coleção trata-se da versão genérica da classe **Hashtable**. Todas as operações fornecidas por esta classe baseiam-se nos tipos especificados em sua declaração. Esta classe fornece em sua construção dois tipos que devemos informar: **TKey** e **TValue**. O primeiro representa o tipo que será a chave e o segundo representará o tipo que será o objeto que devemos aceitar como valor, associado a chave.

Assim como o **Hashtable**, a chave não pode ser uma referência nula, mas o valor pode ser, sem problema algum. Ainda há uma mudança com relação ao objeto que representará a chave e o valor durante a iteração da coleção. Ao invés de utilizarmos a estrutura **DictionaryEntry**, estaremos utilizando a estrutura **KeyValuePair<TKey, TValue>** que é a versão genérica da estrutura anterior e será utilizado por todas as coleções genéricas que operam utilizando chave e valor.

Essa coleção utiliza uma implementação da *Interface* **IEqualityComparer** para determinar se as chaves são ou não iguais. Para isso, você pode especificar essa implementação customizada através do construtor da classe **Dictionary**.

O código abaixo exibe uma uma solução onde a chave deve ser uma *string* e o valor um objeto do tipo *Usuario*. Depois de adicionar os itens e suas respectivas chaves, ele exibe os mesmos através de um laço **For Each**. Se reparar, logo após a propriedade *Key* ou *Value* da estrutura **KeyValuePair**, já podemos acessar qualquer membro dos objetos, como é o caso da propriedade *Nome* da classe *Usuario*:

```
VB.NET
Imports System.Collections.Generic

Dim dic As New Dictionary(Of String, Usuario)
dic.Add("JT", New Usuario("Jose Torres"))
dic.Add("ZC", New Usuario("Zuleika Camargo"))
dic.Add("MA", New Usuario("Maria Antonieta"))

For Each pair As KeyValuePair(Of String, Usuario) In dic
    Console.WriteLine(pair.Value.Nome)
Next
```

```
C#
using System.Collections.Generic;

Dictionary<string, Usuario> dic =
    new Dictionary<string, Usuario>();

dic.Add("JT", new Usuario("Jose Torres"));
dic.Add("ZC", new Usuario("Zuleika Camargo"));
dic.Add("MA", new Usuario("Maria Antonieta"));

foreach (KeyValuePair<string, Usuario> pair in dic)
{
    Console.WriteLine(pair.Value.Nome);
}
```

SortedList<TKey, TValue> e SortedDictionary<TKey, TValue>

Novamente, a **SortedList<TKey, TValue>** e a versão genérica para a classe **SortedList**, que opera com tipos genéricos. Essa classe recebe em seu construtor uma implementação da *Interface* **IComparer** para determinar se as chaves são ou não iguais.

A **SortedDictionary<TKey, TValue>** trata-se de uma nova coleção que foi introduzida na versão 2.0 do .NET Framework, que também trabalha como uma coleção de chave e valor, baseando-se em uma única chave. Assim como a **SortedList**, é também ordenada pela chave, mas com uma importante diferença: é muito mais rápido em comparada a **SortedList<TKey, TValue>** em relação a inserção e remoção de elementos, mas utiliza mais memória que ela.

A utilização destas classes são bem semelhantes. O que realmente muda é o comportamente interno de cada uma delas.

```
VB.NET
Imports System.Collections.Generic

Dim sl As New SortedList(Of String, Usuario)
Dim sd As New SortedDictionary(Of String, Usuario)

sl.Add("MA", New Usuario("Maria Antonieta"))
sd.Add("MA", New Usuario("Maria Antonieta"))

C#
using System.Collections.Generic;

SortedList<string, Usuario> sl = new SortedList<string, Usuario>
();
SortedDictionary<string,       Usuario>       sd       =       new
```

```
SortedDictionary<string, Usuario> ();

sl.Add("MA", new Usuario("Maria Antonieta"));
sd.Add("MA", new Usuario("Maria Antonieta"));
```

LinkedList<T>

Imagine que temos uma lista de tarefas a serem realizadas em um determinado projeto. Essas tarefas são melhorias, validações e também alguns ajustes que devem ser realizados para que o projeto continue funcionando. Você e sua equipe vão desenvolvem a lista com todas essas tarefas e, neste momento, não se preocupam com a propriedade de cada uma delas.

Como tal projeto já está em produção e os clientes estão a todo vapor o utilizando, as melhorias devem ser as últimas a serem realizadas. Já as validações e os ajustes devem ter uma prioriodade maior sobre as melhorias. Neste cenário, como é que podemos adicionar os ajustes e validações acima das melhorias, que não são uma necessidade no momento?

Transformando todo esse cenário em .NET, mais especificamente em uma coleção, essas tarefas, ainda sem uma ordem específica, foram sendo adicionadas à uma coleção. Agora é necessário adicionar as tarefas especificando um nó (tarefa) que é representado pelo elemento da coleção e assim, podemos inserir a nova tarefa antes ou depois do nó especificado durante a inserção do novo elemento.

A coleção que nos oferece essa estrutura é a **LinkedList<T>**. Ela permite inserir um elemento na coleção antes ou depois de elemento já existente que é especificado durante a inserção ou remoção. Cada um dos elementos que são inseridos dentro desta coleção são encapsulados dentro de uma classe do tipo **LinkedListNode<T>**. Como trata-se de uma coleção do tipo "linked", a classe **LinkedListNode<T>** fornece duas propriedades bastante interessantes, chamadas: *Previous* e *Next* que retornam também um objeto do tipo **LinkedListNode<T>**. O primeiro, *Previous*, retorna uma referência para o nó anterior e nulo se estiver sendo chamado através do primeiro nó da coleção. Já o método *Next*, retorna uma referência para o nó seguinte, também retornando nulo se a propriedade estiver sendo chamada através do último nó da coleção. Já entre os membros da classe **LinkedList<T>**, temos alguns que merecem serem citados:

Membro	Descrição
First	Esta propriedade retorna um objeto do tipo **LinkedListNode<T>** que representa o primeiro elemento da coleção. Se a coleção estiver vazia, será retornado um valor nulo.
Last	Esta propriedade retorna um objeto do tipo **LinkedListNode<T>** que representa o último elemento da coleção. Se a coleção estiver vazia, será retornado um valor nulo.

AddAfter	Adiciona um novo nó com o novo valor, logo após o nó especificado.
AddBeforer	Adiciona um novo nó com o novo valor, antes do nó especificado.
AddFirst	Adiciona um novo nó no início da coleção.
AddLast	Adiciona um novo nó no final da coleção.
Find	Dado um valor, retorna a um objeto do tipo **LinkedListNode<T>** que representa o nó em que o objeto especificado está encapsulado. Se a coleção não encontrar o valor especificado, uma referência nula é retornada.
RemoveFirst	Remove o primeiro nó da coleção.
RemoveLast	Remove o último nó da coleção.

Para exemplificar, será criado uma coleão **LinkedList<T>** onde teremos as tarefas de um determinado projeto. Teremos três tarefas de categorias diferentes: melhoria, ajuste e erro. Como os erros são prioridades, devemos sempre colocá-los em uma sequência, já que os erros são sempre críticos e tem uma maior prioridade em relação a qualquer outra tarefa. Sendo assim, analise o código abaixo:

```
VB.NET
Imports System.Collections.Generic

Dim projeto As New LinkedList(Of String)

projeto.AddFirst("[Melhoria]: Melhoria 1.")
projeto.AddFirst("[Ajuste]: Ajuste 1.")
projeto.AddFirst("[Erro]: Erro 1.")

Dim erro1 As LinkedListNode(Of String) = projeto.Find("[Erro]:
Erro 1.")
If Not IsNothing(erro1) Then
    projeto.AddAfter(erro1, "[Erro]: Erro 2.")
End If

For Each node As String In projeto
    Console.WriteLine(node)
Next

C#
using System.Collections.Generic;

LinkedList<string> projeto =
    new LinkedList<string>();

projeto.AddFirst("[Melhoria]: Melhoria 1.");
projeto.AddFirst("[Ajuste]: Ajuste 1.");
projeto.AddFirst("[Erro]: Erro 1.");

LinkedListNode<string> erro1 = projeto.Find("[Erro]: Erro 1.");
```

```
if(erro1 != null)
    projeto.AddAfter(erro1, "[Erro]: Erro 2.");

foreach (string node in projeto)
    Console.WriteLine(node);
```

Inicialmente criamos a instância da coleção **LinkedList** especificando o tipo *string*. Em seguida, adicionamos três itens (tarefas) através do método *AddFirst*. Como este método adiciona o item sempre em primeiro lugar na lista, neste caso, o último será o primeiro. Como no nosso cenário é importante termos os *erros* em primeiro lugar, ao receber um novo *erro* que deve ser colocado no lista, precisamos recuperar o último *erro* inserido através do método *Find* e, se encontrado na coleção, devemos invocar o método *AddAfter* para adicionar essa nova tarefa imediatamente abaixo do última tarefa de *erro* criada.

Para ambos os códigos, o retorno será o mesmo:

```
[Erro]: Erro 1.
[Erro]: Erro 2.
[Ajuste]: Ajuste 1.
[Melhoria]: Melhoria 1.
```

Iterators

Os *Iterators* é uma nova funcionalidade disponibilizada apenas pelo Visual C# 2.0. Logo no início deste capítulo, em Coleções Primárias, vimos que para que possamos iterar entre os elementos de uma coleção customizada através de um laço **foreach** (**For Each** em Visual Basic .NET) é necessário implementarmos a *Interface* **IEnumerable**.

Esta *Interface* **IEnumerable** expõe um método chamado *GetEnumerator* que retorna um enumerador (classe que implementa a *Interface* **IEnumerator**) que, por sua vez, fornecem os seguintes membros: *MoveNext*, *Current*, *Reset* e *Dispose*, quais são utilizados pelo runtime para que ele possa recuperar cada item da coleção. Agora, com os *Iterators*, isso não é mais necessário, já que quando o compilador detecta a presença de um *Iterator*, ele automaticamente gera o código necessário para a criação do enumerador. Com isso, não será mais necessário criarmos uma classe, geralmente uma classe privada, que implemente a *Interface* **IEnumerator** que é retornada através do método *GetEnumerator*.

Para exemplificar o uso de um *Iterator*, podemos utilizar o mesmo exemplo que utilizamos quando foi abordado sobre as *Interfaces* **IEnumerable** e **IEnumerator**. A idéia é iterar entre as categorias definidas no interior de uma classe qualquer:

```
C#
public class Categories : IEnumerable
{
    private string[] _categories =
        new string[ ] { "ASP.NET", "VB.NET", "C#", "SQL", "XML"
};

    public IEnumerator GetEnumerator()
    {
        for (int i = 0; i < this._categories.Length; i++)
        {
            yield return this._categories[i];
        }
    }
}

//Utilização:

foreach (string category in new Categories())
    Console.WriteLine(category);
```

Apesar de não mais precisarmos de uma classe auxiliar para criar o enumerador, quando utilizamos a *keyword* **yield**, automaticamente o compilar gera a classe que implementa a *Interface* **IEnumerator**, implementando todos os métodos necessários para a iteração entre os elementos da coleção. A *keyword* **yield** que determina a criação do enumerador, ainda permite uma outra forma de passar os itens para o mesmo, mantendo o mesmo resultado final. O trecho de código abaixo é mostrado essa outra forma da utilização do *Iterator*:

```
C#
public class Categories : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        yield return "ASP.NET";
        yield return "VB.NET";
        yield return "C#";
        yield return "SQL";
        yield return "XML";
    }
}
```

Nota: Apesar de que o exemplo está sendo mostrado com a *Interface* primária **IEnumerable**, você pode também implementar os *Iterators* com a *Interface* genérica **IEnumerable<T>**.

Coleções Especializadas

Como o próprio nome diz, as coleções especializadas são coleções criadas especificamente para um propósito muito especial. A Microsoft disponibilizou várias delas que estão contidas dentro do *namespace* **System.Collections.Specialized**. Essas classes foram criadas para operarem sempre com um determinado tipo, justamente para conseguir adaptar e otimizar o melhor possível a coleção, sem *overheads* extras.

As coleções especializadas fornecidas pelo .NET Framework estão divididas em quatro categorias, quais podemos analisar através da tabela abaixo:

Categoria	**Descrição**
Strings Classes	São coleções que foram criadas exclusivamente para manipularem *strings*: **StringCollection**, **StringDictionary**, **StringEnumerator** e **CollectionsUtil**.
Dictionary Classes	Coleções contidas dentro desta categoria, fornecem dicionários de dados de algo performance que, dependendo do que armazenam e da quantidade de elementos, mudam o seu comportamento interno para otimizar a leitura, gravação e o armazenamento dos dados. Entre as coleções desta categoria temos: **ListDictionary**, **HybridDictionary** e **OrderedDictionary**.
Named Collection Classes	As classes disponibilizadas aqui fornecem uma base para uma coleção que chaves *string*s e valores *objects* que você pode acessar através da chave ou de um índice. Isso tipo de coleção permitirá que adicionemos valores duplicados, agrupando por uma chave. Como exemplo destas classes temos: **NameValueCollection** e **NameObjectCollectionBase**.
Bit Structures	Fornece uma estrutura para armazenar valores booleanos e pequenos inteiros em 32-bits de memória. Para isso temos as seguintes estruturas: **BitVector32** e **BitVector32.Section**.

Cada uma das categorias e suas respectivas coleções estarão detalhadamente explicadas nas próximas páginas.

Coleções 31

Specialized String Classes

StringCollection

Esta coleção é basicamente análoga a uma coleção do tipo **ArrayList** em termos de funcionalidades. A única diferença é que a classe **StringCollection** trabalha apenas com *strings*. Internamente, a classe **StringCollection** armazena os valores em um objeto do tipo **ArrayList** e, serve de *wrapper* para o mesmo, fornecendo ao cliente todos os métodos necessários para manipular os elementos dentro dele contidos. A sua implementação é tão simples quanto a do **ArrayList**:

VB.NET
```
Imports System.Collections.Specialized

Dim categorias As New StringCollection()
categorias.Add("ASP.NET")
categorias.Add("VB.NET")
categorias.Add("C#")
categorias.Add("XML")

'Utilização:

For index As Integer = 0 To categorias.Count - 1
    Console.WriteLine(categorias(index))
Next
```

C#
```
using System.Collections.Specialized;

StringCollection categorias = new StringCollection();
categorias.Add("ASP.NET");
categorias.Add("VB.NET");
categorias.Add("C#");
categorias.Add("XML");

for (int index = 0; index < categorias.Count; index++)
    Console.WriteLine(categorias[index]);
```

StringDictionary

Esta coleção é basicamente análoga a uma coleção do tipo **Hashtable** em termos de funcionalidades. A única diferença é que a classe **StringDictionary** trabalha apenas com *strings* tanto na chave quanto no valor, podendo apenas o valor ser uma referência nula. Internamente, a classe **StringDictionary** armazena os valores em um objeto do tipo **Hashtable** e, serve de *wrapper* para o mesmo, fornecendo ao cliente todos os métodos necessários para manipular os elementos dentro dele contidos.

Um pequena diferença é que, ao contrário da classe/coleção **Hashtable**, esta coleção trabalha independentemente de maiúsculas e minúsculas, fazendo com que as chaves sejam armazenadas em *lowercase*. A sua implementação é tão simples quanto a do **Hashtable**:

```
VB.NET
Imports System.Collections.Specialized

Dim dic As new StringDictionary()
dic.Add("AC", "Acre")
dic.Add("RJ", "Rio de Janeiro")
dic.Add("SP", "São Paulo")

For Each entry As DictionaryEntry In dic
    Console.WriteLine(entry.Key & " - " & entry.Value)
Next

C#
using System.Collections.Specialized;

StringDictionary dic = new StringDictionary();
dic.Add("AC", "Acre");
dic.Add("RJ", "Rio de Janeiro");
dic.Add("SP", "São Paulo");

foreach (DictionaryEntry entry in dic)
    Console.WriteLine(entry.Key + " - " + entry.Value);
```

StringEnumerator

Como já vimos acima, os laços **foreach** (**For Each** em Visual Basic .NET) ocultam toda a complexidade dos enumeradores, e além disso, é recomendado pela Microsoft ao invés de manipular diretamente o enumerador. Para saber mais sobre os enumeradores, consulta a seção das coleções primárias onde são abordadas as *Interfaces* **IEnumerable** e **IEnumerator**.

A classe **StringEnumerator** é um enumerador que manipula uma coleção de *strings*. Com isso, o método *GetEnumerator* da classe **StringCollection** retorna um enumerador do tipo **StringEnumerator**, qual utilizaremos para iterar entre os elementos da coleção. O código abaixo exibe um exemplo de como extrair o enumerador de uma **StringCoolection**:


```
VB.NET
Imports System.Collections.Specialized
```

```
Dim cats As New StringCollection()
cats.AddRange(New String() {"ASP.NET", "VB.NET", "C#", "SQL",
"XML"})

Dim enumerador As StringEnumerator = cats.GetEnumerator()
While (enumerador.MoveNext())
    Console.WriteLine(enumerador.Current)
End While
```

C#

```
using System.Collections.Specialized;

StringCollection cats = new StringCollection();
cats.AddRange(new string[] { "ASP.NET", "VB.NET", "C#", "SQL",
"XML" });

StringEnumerator enumerador = cats.GetEnumerator();
while (enumerador.MoveNext())
    Console.WriteLine(enumerador.Current);
```

CollectionsUtil

Esta classe fornece dois métodos estáticos para a criação de coleções que ignoram a diferenciação entre maiúsculas e minúsculas. Ela fornece dois que auxiliam na criação das coleções: *CreateCaseInsensitiveHashtable* e *CreateCaseInsensitiveSortedList*. O primeiro deles, retorna uma instância de um Hashtable que não fará a distinção entre maiúsculas e minúsculas, o que significa que a chave "SP" e "sp" são iguais. Já o segundo método, *CreateCaseInsensitiveSortedList*, tem a mesma finalidade, só que retorna um objeto do tipo SortedList que ignora também maiúsculas e minúsculas. O código abaixo exibe a criação das coleções acima citadas através dos métodos estáticos fornecidos pela classe **CollectionsUtil**:

VB.NET

```
Imports System.Collections.Specialized

Dim t As Hashtable = _
    CollectionsUtil.CreateCaseInsensitiveHashtable()

Dim s As SortedList = _
    CollectionsUtil.CreateCaseInsensitiveSortedList()
```

C#

```
using System.Collections.Specialized;

Hashtable hashTable =
    CollectionsUtil.CreateCaseInsensitiveHashtable();
```

```
SortedList sortedList =
    CollectionsUtil.CreateCaseInsensitiveSortedList();
```

Uma outra alternativa em relação a classe **CollectionsUtil** é passar como parâmetro para as classes **Hashtable** ou **SortedList** uma instância da classe **StringComparer** que foi implementanda para ignorar a diferenciação entre maiúsculas e minúsculas. Um dos overloads do construtor dessas classes recebem um tipo **IEqualityComparer**, onde você pode informar a implementação que é fornecida através do método estático **CurrentCultureIgnoreCase** da classe **StringComparer**. A alternativa é exibida através do código abaixo:

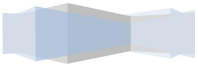
```
VB.NET
Imports System.Collections.Specialized

Dim t As Hashtable = _
    New Hashtable(StringComparer.CurrentCultureIgnoreCase)

Dim s As SortedList = _
    New SortedList(StringComparer.CurrentCultureIgnoreCase)

C#
using System.Collections.Specialized;

Hashtable hashTable =
    new Hashtable(StringComparer.CurrentCultureIgnoreCase);

SortedList lisortedListst =
    new SortedList(StringComparer.CurrentCultureIgnoreCase);
```

Specialized Dictionary Classes

ListDictionary

Este dicionário de dados é uma implementação simples da *Interface* **IDictionary**. Ele é muito menor e mais rápido quando comparado com o **Hashtable** e se o número de elemento for menor ou igual a 10 e, não deve ser utilizado se a performance é importante para uma coleção com muitos elementos.

Os itens que estão dentro deste dicionário não garantem que estarão em uma ordem específica e o código que você escreve não deve depender da ordem corrente de adição. A sua utilização em termos de escrita de código é idêntica ao **Hashtable**. É necessário instanciá-lo e, através do método *Add*, adicionar a chave e valor.



HybridDictionary

Essa coleção tem um comportamente um tanto quanto especial. Trata-se também de um dicionário de dados que implementa a *Interface* **IDictionary** e que, internamente, armazena os dados em um objeto do tipo **ListDictionary** enquanto a coleção é pequena e, quando a coleção ganha um grande número de elemento, automaticamente o repositório é trocado por uma **Hashtable**.

Com este comportamento, esta coleção é recomendada para casos onde o número de elementos é desconhecido, onde ele garantirá a performance enquanto a coleção estiver pequena e também para quando esta mesma coleção ganhar uma proporção maior. Além disso, essa coleção recebe em seu construtor um valor booleano indicando se a coleção vai ou não ignorar a diferença entre maiúsculas e minúsculas quando for comparar uma chave.

Mais uma vez, a utilização da coleção **HybridDictionary** em termos de escrita de código é idêntica ao **Hashtable**. É necessário instanciá-lo e, através do método *Add*, adicionar a chave e valor.

OrderedDictionary

Essa classe implementa a *Interface* **IOrderedDictionary** que representa uma coleção indexada através de pares de chave e valor. Toda coleção que implementa essa *Interface* pode acessar seus elementos através de um chave ou índice. É basicamente uma forma de combinar um **ArrayList** com um **Hashtable**, pois os elementos do **ArrayList** somente são acessíveis através de índices e o **Hashtable**, podemos acessar os elementos através de uma chave.

Como a *Interface* **IOrderedDictionary** também fornece uma propriedade *default* (*indexer* em Visual C#), temos agora duas propriedades deste tipo para recuperar um determinado elemento. A diferença entre as duas propriedade é que uma delas receberá um número inteiro que representará o elemento que desejamos recuperar; a segunda, receberá um objeto que representa a chave que foi utilizada para inserir o elemento dentro da coleção. Podemos visualizar isso através do código abaixo:

```
VB.NET
Imports System.Collections.Specialized

Dim dic As New OrderedDictionary
dic.Add("AC", "Acre")
dic.Add("RJ", "Rio de Janeiro")
dic.Add("SP", "São Paulo")

Console.WriteLine(dic("RJ"))
Console.WriteLine(dic(1))
```

```
C#
using System.Collections.Specialized;

OrderedDictionary dic = new OrderedDictionary();
dic.Add("AC", "Acre");
dic.Add("RJ", "Rio de Janeiro");
dic.Add("SP", "São Paulo");

Console.WriteLine(dic["RJ"]);
Console.WriteLine(dic[1]);
```

Ambas as formas (através da chave e através do índice) retornarão "Rio de Janeiro".

Specialized Named Collection Classes

NameObjectCollectionBase

Esta é uma classe abstrata para as coleções considerada "*named collections*". Internamente ela mantém um objeto do tipo **Hashtable** onde sua chave é uma *string* e seu valor um *object*. Como trata-se de uma classe abstrata, você pode estar customizando a sua própria coleção a partir desta classe base.

Quando for criar a sua própria coleção herdando desta, não se preocupe em precisar implementar nenhum método. Apenas alguns métodos são denotados como *protected virtual* (*Protected Overridable* em Visual Basic .NET), quais você pode customizar para a sua coleção.

NameValueCollection

Derivada de **NameObjectCollectionBase**, esta é uma coleção que pode ser acessada através de uma chave ou um índice. Mas o diferencial desta classe é mesmo a possibilidade de armazenar múltiplas valores (strings) através de uma única chave. Isso quer dizer que você pode adicionar chaves repetidas que, internamente, a coleção se encarrega de ajustar para quando desejar extrair os valores, todos eles sejam retornados.

Alguns exemplos típicos deste tipo de coleção são as propriedades *QueryString*, *Forms* e *Headers* da classe **HttpRequest**, utilizada em aplicações ASP.NET. Para exemplificar a utilização desta coleção, analise o código abaixo:

```
VB.NET
Imports System.Collections.Specialized

Dim queryStrings As New NameValueCollection
queryStrings.Add("Alunos", "José")
queryStrings.Add("Alunos", "Ivan")
```

```
queryStrings.Add("Alunos", "Mario")
queryStrings.Add("Alunos", "Castro")
queryStrings.Add("Alunos", "Mendes")
queryStrings.Add("Instrutor", "Israel")

For Each s As String In queryStrings.AllKeys
    Console.WriteLine("{0} - {1}", s, queryStrings(s))
Next
```

C#

```
using System.Collections.Specialized;

NameValueCollection queryStrings = new NameValueCollection();
queryStrings.Add("Alunos", "José");
queryStrings.Add("Alunos", "Ivan");
queryStrings.Add("Alunos", "Mario");
queryStrings.Add("Alunos", "Castro");
queryStrings.Add("Alunos", "Mendes");
queryStrings.Add("Instrutor", "Israel");

foreach (string s in queryStrings.AllKeys)
    Console.WriteLine("{0} - {1}", s, queryStrings[s]);
```

Como podemos visualizar, é perfeitamente permitido adicionar chaves iguais que a coleção trabalha sem nenhum problema. Em seguinda, dentro do laço percorremos a coleção distinta de chaves e, através de uma propriedade *default* (*indexer* em Visual C#) da classe **NameValueCollection**, passamos a chave corrente e ela retornará todos os valores que estão vinculados a chave informada. Para ambos os código, o retorno será o seguinte:

```
Alunos - José,Ivan,Mario,Castro,Mendes
Instrutor – Israel
```

Bit Structures

BitVector32 e BitVector32.Section

Essas duas estruturas trabalham em conjunto e permitem você armazenar dentro dela valores inteiros ou valores booleanos utilizando apenas 32 bits de memória. Essa estrutura tem uma performance superior quando comparado a classe **BitArray**, justamente por manipular tipos-valor e não tipos-referência.

Comparer

Ambos namespaces **System.Collections** e **System.Collections.Generic** possuem classes com implementações padrões das *Interfaces* **IComparer** e **IComparer<T>**, respectivamente. Os dois tópicos abaixo explicam e ilustram a utilização de cada uma dessas classes.

Comparer Primário (System.Collections)

Dentro deste *namespace* temos uma classe chamada **Comparer**m que possui uma implementação básica da *Interface* **IComparer** (discutida no capítulo 1). Essa classe recebe em seu construtor um objeto do tipo **CultureInfo** (contido no *namespace* **System.Globalization**) utilizando isso para a globalização da aplicação, pois em diferentes culturas, podemos ter diferentes formas de comparação e também de resultados. O exemplo abaixo ilustra como utilizá-la, especificando a nossa cultura ("pt-BR") para efetuar a comparação:

```
VB.NET
Imports System.Collections
Imports System.Globalization

Dim comp As New Comparer(New CultureInfo("pt-BR"))
Console.WriteLine("Comparando Paulo e paulo : {0}", _
    comp.Compare("Paulo", "paulo"))

C#
using System.Collections;
using System.Globalization;

Comparer comp = new Comparer(new CultureInfo("pt-BR"));
Console.WriteLine("Comparando Paulo e paulo : {0}",
    comp.Compare("Paulo", "paulo"));
```

Como já sabemos, o método *Compare* retorna um número inteiro indicando se um objeto é menor, igual ou maior que o outro. No nosso caso, será retornado 1, indicando que "Paulo" é maior que "paulo".

Essa classe ainda possui um membro público estático chamado *Default*, que retorna uma instância da classe **Comparer** com a cultura especificada dentro da *thread* atual (**Thread.CurrentCulture**).

Comparer Genérico (System.Collections.Generic)

Quando necessitamos ordenar uma lista, como por exemplo, a classe **List<T>**, necessitamos que os objetos contidos nela sejam ordenados alfabéticamente. Sendo assim, quando você chama o método *Sort* da classe **List<T>** e o seu objeto não

implementar a *Interface* genérica **IComparable<T>** ou a *Interface* primária **IComparable**, uma excessão é disparada.

Como trata-se de uma classe genérica chamada **Comparer<T>**. O fato dela ser genérica, é necessário especificarmos na sua chamada, o tipo com o qual ela irá trabalhar. Para o nosso exemplo, desejaremos ordenar uma lista, que contém objetos do tipo *Usuario*, através do nome (*string*), de forma alfabética. Essa classe fornece uma propriedade estática, de somente leitura chamada *Default*, que retorna um *comparer* específico para o tipo informado que, no nosso caso, será uma *string*. Para o exemplo, foi construído uma classe chamada *Usuario* que, em seu construtor, é passado o nome do mesmo e, expõe uma propriedade de escrita/leitura chamada *Nome*. O trecho de código abaixo exibe na íntegra a construção da classe e também o código responsável pela ordenação da lista de usuários:

```
VB.NET
Public Class Usuario
    Private _nome As String

    Public Sub New(ByVal nome As String)
        Me._nome = nome
    End Sub

    Public Property Nome() As String
        Get
            Return Me._nome
        End Get
        Set(ByVal value As String)
            Me._nome = value
        End Set
    End Property
End Class

'...

Imports System.Collections.Generic

Sub Main()
    Dim l As New List(Of Usuario)
    l.Add(New Usuario("Israel"))
    l.Add(New Usuario("Roberto"))
    l.Add(New Usuario("Anderson"))
    l.Add(New Usuario("Paulo"))
    l.Add(New Usuario("Leandro"))
    l.Sort(New Comparison(Of Usuario)(AddressOf InternalSort))

    For Each u As Usuario In l
        Console.WriteLine(u.Nome)
    Next
```

```
End Sub

Private Function InternalSort(ByVal u1 As Usuario, ByVal u2 As
Usuario) As Integer
    Return Comparer(Of String).Default.Compare(u1.Nome, u2.Nome)
End Function
```

C#

```
public class Usuario
{
    private string _nome;

    public Usuario(string nome)
    {
        this._nome = nome;
    }

    public string Nome
    {
        get
        {
            return this._nome;
        }
        set
        {
            this._nome = value;
        }
    }
}

//...

using System.Collections.Generic;

static void Main(string[] args)
{
    List<Usuario> l = new List<Usuario>();
    l.Add(new Usuario("Israel"));
    l.Add(new Usuario("Roberto"));
    l.Add(new Usuario("Anderson"));
    l.Add(new Usuario("Paulo"));
    l.Add(new Usuario("Leandro"));
    l.Sort(new Comparison<Usuario>(InternalSort));

    foreach (Usuario u in l)
        Console.WriteLine(u.Nome);
}

private static int InternalSort(Usuario u1, Usuario u2)
{
```

```
    return Comparer<string>.Default.Compare(u1.Nome, u2.Nome);
}
```

Nota: Lembrando que no Visual C# há a possibilidade de utlizarmos os *métodos anônimos* para evitar a criação de um procedimento adicional.