

# Developer's Guide to Collections in Microsoft® .NET



**Calvin Janes**

# Developer's Guide to Collections in Microsoft® .NET

## Build the skills to apply Microsoft .NET collections effectively

Put .NET collections to work—and manage issues with GUI data binding, threading, data querying, and storage. Led by a data collection expert, you'll gain task-oriented guidance, exercises, and extensive code samples to tackle common problems and improve application performance. This one-stop reference is designed for experienced Microsoft Visual Basic® and C# developers—whether you're already using collections or just starting out.

### Discover how to:

- Implement arrays, associative arrays, stacks, linked lists, and other collection types
- Apply built-in .NET collection classes by learning their methods and properties
- Add enumerator, dictionary, and other .NET collection interfaces to your classes
- Query collections by writing simple to complex Microsoft LINQ statements
- Synchronize data across threads using built-in .NET synchronization classes
- Enhance your custom collection classes with serialization support
- Use simple data binding to display collections in Windows® Forms, Microsoft Silverlight®, and Windows Presentation Foundation



### About the Author

**Calvin Jones**, a principal software consultant with more than 20 years of professional experience, is an expert in creating and using collection types. He has put his expertise to work on developing everything from 3D graphics to software for helicopters and control systems.

### RESOURCE ROADMAP

#### Developer Step by Step

- Hands-on tutorial covering fundamental techniques and features
- Practice exercises
- Prepares and informs new-to-topic programmers



#### Developer Reference

- Expert coverage of core topics
- Extensive, pragmatic coding examples
- Builds professional-level proficiency with a Microsoft technology



#### Focused Topics

- Deep coverage of advanced techniques and capabilities
- Extensive, adaptable coding examples
- Promotes full mastery of a Microsoft technology



*See inside cover*

### Get code and project samples on the web

Ready to download at

<http://go.microsoft.com/fwlink/?LinkId=227007>

For **system requirements**, see the *Introduction*.

ISBN: 978-0-7356-5927-8



[microsoft.com/mspress](http://microsoft.com/mspress)

**U.S.A. \$34.99**

Canada \$36.99

[Recommended]

Programming/Microsoft .NET

Microsoft®  
Visual Studio®

**Microsoft®**

[www.it-ebooks.info](http://www.it-ebooks.info)

**Microsoft**

# Developer's Guide to Collections in Microsoft® .NET

*Calvin Janes*

Published with the authorization of Microsoft Corporation by:  
O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, California 95472

Copyright © 2011 by Calvin Janes

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-5927-8

1 2 3 4 5 6 7 8 9 LSI 6 5 4 3 2 1

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at [mspininput@microsoft.com](mailto:mspininput@microsoft.com). Please tell us what you think of this book at <http://www.microsoft.com/learning/booksurvey>.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, O'Reilly Media, Inc., Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

**Acquisitions and Developmental Editor:** Russell Jones

**Production Editor:** Holly Bauer

**Editorial Production:** Online Training Solutions, Inc.

**Technical Reviewer:** Chris G. Williams

**Copyeditor:** Jaime Odell, Online Training Solutions, Inc.

**Proofreader:** Victoria Thulman, Online Training Solutions, Inc.

**Indexer:** Ron Strauss

**Cover Design:** Twist Creative • Seattle

**Cover Composition:** Karen Montgomery

**Illustrator:** Jeanne Craver, Online Training Solutions, Inc.

# Contents at a Glance

## Part I Collection Basics

1	Understanding Collections: Arrays and Linked Lists .....	3
2	Understanding Collections: Associative Arrays .....	87
3	Understanding Collections: Queues, Stacks, and Circular Buffers .....	153

## Part II .NET Built-in Collections

4	Generic Collections.....	215
5	Generic and Support Collections .....	283

## Part III Using Collections

6	.NET Collection Interfaces .....	345
7	Introduction to LINQ .....	441
8	Using Threads with Collections.....	469
9	Serializing Collections .....	513

## Part IV Using Collections with UI Controls

10	Using Collections with Windows Form Controls .....	539
11	Using Collections with WPF and Silverlight Controls .....	583



# Table of Contents

Introduction .....	xiii
<b>Part I Collection Basics</b>	
<b>1 Understanding Collections: Arrays and Linked Lists .....</b>	<b>3</b>
Array Overview .....	3
Uses of Arrays .....	3
Advantages of Arrays .....	4
Disadvantages of Arrays .....	4
Array Implementation .....	5
Understanding Indexing .....	5
Getting Started .....	5
Creating Constructors .....	8
Allowing Users to Add Items .....	10
Allowing Users to Remove Items .....	13
Adding Helper Methods and Properties .....	17
Using the <i>ArrayEx(T)</i> Class .....	23
Linked List Overview .....	27
Uses of Linked Lists .....	27
Advantages of Linked Lists .....	27
Disadvantages of Linked Lists .....	28
Linked List Implementation .....	28
Singly Linked List Implementation .....	28
Doubly Linked List Implementation .....	54
Using an Array to Create a Linked List .....	81
Using the Linked List Class .....	81
Summary .....	85

---

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

[microsoft.com/learning/booksurvey](http://microsoft.com/learning/booksurvey)

<b>2 Understanding Collections: Associative Arrays</b>	<b>87</b>
Associative Array Overview	87
Uses of Associative Arrays	87
Advantages and Disadvantages of Associative Arrays	88
Associative Array Implementation	88
Using Association Lists for Associative Arrays	88
Using Hash Tables for Associative Arrays	105
Using the Associative Array Class	147
Summary	152
<b>3 Understanding Collections: Queues, Stacks, and Circular Buffers</b>	<b>153</b>
Queue Overview	153
Uses of Queues	153
Queue Implementation	153
Using an Array to Implement a Queue	154
Using a Linked List to Implement a Queue	163
Using the Queue Classes	170
Stack Overview	175
Uses of Stacks	175
Stack Implementation	175
Adding the Common Functionality	175
Using an Array to Implement a Stack	177
Using a Linked List to Implement a Stack	182
Using the Stack Classes	187
Circular Buffer Overview	193
Uses of Circular Buffers	193
Circular Buffer Implementation	194
Getting Started	194
Understanding the Internal Storage	195
Creating Constructors	198
Allowing Users to Add Items	201
Allowing Users to Remove Items	203
Adding Helper Methods and Properties	205
Changing Capacity	207
Using the <i>CircularBuffer(T)</i> Class	210
Summary	211

## Part II .NET Built-in Collections

<b>4 Generic Collections . . . . .</b>	<b>215</b>
Understanding the Equality and Ordering Comparers . . . . .	215
Understanding the Equality Comparer . . . . .	215
Understanding the Ordering Comparer . . . . .	218
Understanding Delegates, Anonymous Methods, and Lambda Expressions . . . . .	219
Lambda Expressions in Visual Basic . . . . .	221
List(T) Overview . . . . .	222
Using the List(T) Class . . . . .	222
Creating a List(T) . . . . .	222
Appending Items to a List(T) . . . . .	223
Traversing a List(T) . . . . .	224
Removing Items from a List(T) . . . . .	228
Inserting Items into a List(T) . . . . .	230
Sorting a List(T) . . . . .	231
Searching a List(T) . . . . .	247
Checking the Contents of a List . . . . .	263
Modifying the List . . . . .	269
LinkedList(T) Overview . . . . .	272
Using the LinkedList(T) Class . . . . .	273
Creating a New LinkedList(T) . . . . .	273
Adding to the LinkedList(T) . . . . .	273
Removing Nodes from the LinkedList(T) . . . . .	277
Obtaining Information on the LinkedList(T) . . . . .	279
Summary . . . . .	281
<b>5 Generic and Support Collections . . . . .</b>	<b>283</b>
Queue(T) Overview . . . . .	283
Using the Queue(T) Class . . . . .	283
Creating a Queue . . . . .	284
Adding Items to the Queue . . . . .	285
Removing Items from the Queue . . . . .	285
Checking the Queue . . . . .	286
Cleaning the Queue . . . . .	287
Stack(T) Overview . . . . .	288

Using the <i>Stack(T)</i> Class .....	288
Creating a Stack.....	288
Adding Items to the Stack.....	289
Removing Items from the Stack.....	290
Checking the Stack.....	291
Cleaning the Stack .....	292
<i>Dictionary(TKey,TValue)</i> Overview .....	292
Understanding <i>Dictionary(TKey,TValue)</i> Implementation.....	293
Using the <i>Dictionary(TKey,TValue)</i> Class.....	293
Creating a Dictionary .....	294
Adding Items to a Dictionary .....	297
Removing Items from a Dictionary .....	298
Retrieving Values from the Dictionary by Using a Key.....	299
Checking the Dictionary .....	301
<i>BitArray</i> Overview.....	306
Using the <i>BitArray</i> Class.....	306
Creating a <i>BitArray</i> .....	306
Accessing Bits in the <i>BitArray</i> Class .....	308
Using the <i>BitArray</i> Class for Bit Operations .....	310
<i>CollectionBase</i> and <i>DictionaryBase</i> Overview .....	316
Using <i>CollectionBase</i> .....	317
Using <i>DictionaryBase</i> .....	324
<i>HashSet(T)</i> Overview .....	329
Using the <i>HashSet(T)</i> Class .....	329
Creating a <i>HashSet(T)</i> .....	329
Adding Items to the <i>HashSet(T)</i> .....	330
Removing Items from a <i>HashSet(T)</i> .....	331
Performing Set Operations on a <i>HashSet(T)</i> .....	333
Sorted Collections Overview.....	339
<i>SortedList(TKey, TValue)</i> .....	339
<i>SortedDictionary(TKey, TValue)</i> .....	339
Summary.....	341

## Part III Using Collections

<b>6 .NET Collection Interfaces . . . . .</b>	<b>345</b>
<b>Enumerators (<i>IEnumerable</i> and <i>IEnumerator</i>) Overview . . . . .</b>	<b>345</b>
<b>Adding Enumeration Support to Classes . . . . .</b>	<b>349</b>
<b><i>ArrayEx(T)</i> . . . . .</b>	<b>349</b>
<b><i>CircularBuffer(T)</i> . . . . .</b>	<b>355</b>
<b><i>SingleLinkedList(T)</i> and <i>DoubleLinkedList(T)</i> . . . . .</b>	<b>360</b>
<b><i>QueuedArray(T)</i> . . . . .</b>	<b>367</b>
<b><i>QueuedLinkedList(T)</i> . . . . .</b>	<b>373</b>
<b><i>StackedArray(T)</i> . . . . .</b>	<b>374</b>
<b><i>StackedLinkedList(T)</i> . . . . .</b>	<b>379</b>
<b><i>AssociativeArrayAL(TKey,TValue)</i> . . . . .</b>	<b>385</b>
<b><i>AssociativeArrayHT(TKey,TValue)</i> . . . . .</b>	<b>391</b>
<b><i>ICollection</i> and <i>ICollection(T)</i> Overview . . . . .</b>	<b>397</b>
<b>Adding Collection Support to Classes . . . . .</b>	<b>398</b>
<b><i>ArrayEx(T)</i> . . . . .</b>	<b>398</b>
<b><i>CircularBuffer(T)</i> . . . . .</b>	<b>401</b>
<b><i>SingleLinkedList(T)</i> and <i>DoubleLinkedList(T)</i> . . . . .</b>	<b>403</b>
<b><i>QueuedArray(T)</i> . . . . .</b>	<b>408</b>
<b><i>QueuedLinkedList(T)</i> . . . . .</b>	<b>411</b>
<b><i>StackedArray(T)</i> . . . . .</b>	<b>412</b>
<b><i>StackedLinkedList(T)</i> . . . . .</b>	<b>415</b>
<b><i>AssociativeArrayAL(TKey,TValue)</i> . . . . .</b>	<b>417</b>
<b><i>AssociativeArrayHT(TKey,TValue)</i> . . . . .</b>	<b>422</b>
<b><i>IList</i> and <i>IList(T)</i> Overview . . . . .</b>	<b>428</b>
<b>Adding <i>IList(T)</i> and <i>IList</i> Support to Classes . . . . .</b>	<b>429</b>
<b><i>ArrayEx(T)</i> . . . . .</b>	<b>429</b>
<b><i>IDictionary(TKey,TValue)</i> Overview . . . . .</b>	<b>434</b>
<b>Adding Key/Value Pair Support to Classes . . . . .</b>	<b>435</b>
<b><i>AssociativeArrayAL(TKey,TValue)</i> . . . . .</b>	<b>436</b>
<b><i>AssociativeArrayHT(TKey,TValue)</i> . . . . .</b>	<b>437</b>
<b>Summary . . . . .</b>	<b>439</b>

<b>7</b>	<b>Introduction to LINQ . . . . .</b>	<b>441</b>
	What Is LINQ? . . . . .	441
	LINQ Basics. . . . .	442
	Potential LINQ Data Sources . . . . .	442
	What You Should Know About Query Execution . . . . .	443
	Getting Started with LINQ. . . . .	449
	Additions to the .NET Language for LINQ . . . . .	451
	Picking a Data Source ( <i>from</i> Clause) and Selecting Results ( <i>select</i> Clause). . . . .	453
	Filtering Results (the <i>where</i> Clause) . . . . .	458
	Ordering Results (the <i>orderby</i> Clause) . . . . .	460
	The <i>group</i> Clause. . . . .	461
	The <i>join</i> Clause. . . . .	463
	The <i>let</i> Clause. . . . .	466
	Summary. . . . .	468
<b>8</b>	<b>Using Threads with Collections. . . . .</b>	<b>469</b>
	What Is a Thread? . . . . .	469
	What Is Thread Synchronization?. . . . .	470
	Why Should I Care About Thread Synchronization? . . . . .	470
	Why Not Write Thread Synchronization Code As Needed? . . . . .	474
	.NET Framework Tools for Synchronization . . . . .	475
	Interlocked Operations . . . . .	475
	Signaling. . . . .	476
	Locking . . . . .	478
	Adding Synchronization Support to Your Collection Classes . . . . .	480
	<code>ICollection</code> Revisited . . . . .	481
	Getting Started . . . . .	481
	<code>SyncRoot</code> vs. the Synchronized Wrapper Class ( <i>IsSynchronized</i> ). . . . .	483
	Using the <i>Monitor</i> Class. . . . .	487
	Using the <i>ReaderWriterLockSlim</i> Class . . . . .	490
	Handling Recursion. . . . .	500
	Using Upgradeable Locks . . . . .	500
	Implementing a Synchronized Wrapper Class . . . . .	504
	Handling Collection Changes While Enumerating . . . . .	511
	Synchronized Collection Classes. . . . .	511
	<code>SynchronizedCollection(T)</code> . . . . .	511

<i>SynchronizedKeyedCollection(T)</i> . . . . .	512
<i>SynchronizedReadOnlyCollection(T)</i> . . . . .	512
Summary. . . . .	512
<b>9 Serializing Collections . . . . .</b>	<b>513</b>
Serialization . . . . .	513
Using the Serializer Formatters . . . . .	513
Applying the <i>Serializable</i> Attribute . . . . .	513
Controlling Serialization Behavior . . . . .	517
Adding Serialization Support to Collection Classes . . . . .	521
The <i>ArrayEx(T)</i> Class . . . . .	522
The Linked List Classes . . . . .	524
The Associative Array Classes . . . . .	528
The Queue Classes . . . . .	532
The Stack Classes . . . . .	533
Summary. . . . .	535
<b>Part IV Using Collections with UI Controls</b>	
<b>10 Using Collections with Windows Form Controls . . . . .</b>	<b>539</b>
Simple Binding. . . . .	539
Two-Way Data Binding. . . . .	540
Implementing the <i>IBindingList</i> Interface . . . . .	541
Implementing the <i>IBindingListView</i> Interface . . . . .	562
Using the <i>BindingList(T)</i> Class . . . . .	574
Using the <i>BindingSource</i> Class . . . . .	575
Understanding the Sample Code. . . . .	576
Binding with the <i>ComboBox</i> Control . . . . .	576
Binding with the <i>ListBox</i> Control . . . . .	577
Binding with the <i>DataGridView</i> Control and <i>IBindingList</i> . . . . .	578
Binding with the <i>DataGridView</i> Control and <i>IBindingListView</i> . . . . .	580
Binding with the <i>BindingSource</i> Object . . . . .	581
Summary. . . . .	582

---

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

[microsoft.com/learning/booksurvey](http://microsoft.com/learning/booksurvey)

<b>11 Using Collections with WPF and Silverlight Controls . . . . .</b>	<b>583</b>
<i>INotifyCollectionChanged</i> Overview . . . . .	583
Implementing the <i>INotifyCollectionChanged</i> Interface . . . . .	583
Notifying the User of Cleared Items . . . . .	590
Resolving Problems with the Recursive Collection Change Event . . . . .	591
<i>ObservableCollection(T)</i> Overview . . . . .	593
Using the <i>ObservableCollection(T)</i> Class . . . . .	593
Handling Recursive Collection Change Events . . . . .	595
<i>ICollectionView</i> and <i>CollectionView</i> Overview . . . . .	596
When to Use <i>BindingListCollectionView</i> . . . . .	597
When to Use <i>ListCollectionView</i> . . . . .	598
Understanding the Sample Code . . . . .	598
Binding with the <i>ComboBox</i> Control . . . . .	598
Binding with the <i>ListBox</i> Control . . . . .	600
Binding with the <i>ListView</i> Control . . . . .	601
Binding with the <i>TreeView</i> Control . . . . .	603
Binding with the <i>CollectionView</i> Class . . . . .	605
Summary . . . . .	609
<b>Index . . . . .</b>	<b>611</b>

# Introduction

Applications that don't internally use some type of collection are difficult to find. Also difficult to find are books about collections. Many of the methods and properties of the Microsoft .NET collections are well documented, but it often seems like you can't find help for the particular one you are interested in. At times, it may feel like everyone in the world should know what a hash table is, how to use it in a multithreaded environment, and when to use a list instead. But when you happen to be one of the unfortunate developers busily searching the Internet for information about how to solve a critical collection problem for tomorrow's application release, you may find the Internet full of inconsistent information. Or you may find yourself throwing figurative duct tape and bandages on threading and performance issues to hold to the release schedule. All of this—and more—was my motivation for creating this book. I wanted to create a one-stop shop for anyone struggling with collections: from beginners to experts who just need a reference or a few pointers here and there. Throughout the book are many useful tips and tricks that you can use with collections. Some you may already know, and others will be new to you. In either case, I hope you enjoy reading it as much as I enjoyed writing it.

## Who Should Read This Book

This book exists to help existing Microsoft Visual Basic and Microsoft Visual C# developers understand collections in .NET. It is useful both for developers designing new applications and developers maintaining existing applications. The book is arranged so that developers who are new to collections can get started quickly, and those who are already familiar with collections can treat the book as a useful reference.

Developers who are not developing in .NET may find the book useful as well.

## Assumptions

This book expects that you have at least a minimal understanding of .NET development and object-oriented programming concepts. Although collections are used in .NET, the majority of the book provides examples in C# and Visual Basic .NET only. If you have not yet learned one of those languages, you might consider reading John Sharp's book *Microsoft Visual C# 2010 Step by Step* (Microsoft Press, 2010) or Michael Halvorson's book *Microsoft Visual Basic 2010 Step by Step* (Microsoft Press, 2010).

The first two chapters cover basic collection types and concepts and assume that you may want to use collections in languages other than .NET. Some care is taken to allow you to do so. Later chapters focus more on .NET.

## Who Should Not Read This Book

This book is not a .NET primer for beginners; it's intended for developers already conversant with .NET and comfortable with either the C# or Visual Basic .NET language.

## Organization of This Book

This book is divided into four parts.

### Part I, Collection Basics

Part I includes Chapters 1 through 3 and introduces you to collections and how to use them. In these chapters, you learn how to implement and use arrays, linked lists, associative arrays, queues, stacks, and circular buffers. You also learn some of the different names for collection classes among developers and in different languages. In these chapters, you start with an empty class, and build it to a fully functional collection class.

### Part II, .NET Built-in Collections

Part II includes Chapters 4 and 5 and introduces you to some of the built-in .NET collection classes. In these chapters, you learn the methods and properties of *List(T)*, *LinkedList(T)*, *Queue(T)*, *Stack(T)*, *Dictionary(TKey, TValue)*, *HashSet(T)*, *BitArray*, *CollectionBase*, *DictionaryBase*, *SortedList(TKey, TValue)*, and *SortedDictionary(TKey, TValue)* and how to use them.

### Part III, Using Collections

Part III includes Chapters 6 through 9 and covers some of the many ways to use collections with different technologies and strategies such as the .NET Framework, Language Integrated Query (LINQ), threading, and serialization.

In Chapter 6, you learn about the .NET interfaces, such as *IEnumerable(T)*, *IEnumerator(T)*, *ICollection(T)*, *IList(T)*, and *IDictionary(TKey, TValue)*. You learn how to implement them in your classes and how to use them.

In Chapter 7, you are introduced to the LINQ. You learn how to write simple and complex LINQ statements and use them with your custom collection classes and the built-in .NET collection classes.

In Chapter 8, you learn about threads and the importance of synchronizing data used across threads. You also learn how to implement synchronization for your custom collection classes and how to use some of the built-in .NET synchronization classes.

In Chapter 9, you learn how to serialize collections and how to add serialization support to your custom collection classes. You also learn how to control what gets serialized in your custom collection classes.

## Part IV, Using Collections with UI Controls

At some point in your development career, you will find a need to display your collection to the end user. Part IV includes Chapters 10 and 11, which show you how to use collections with simple data binding in Windows Forms, Silverlight, and Windows Presentation Foundation (WPF).

## Finding Your Best Starting Point in This Book

*Developer's Guide to Collections in Microsoft .NET* is designed to be a complete guide to collections. Depending on your needs and your existing understanding of collections, you may want to focus on specific areas of the book. Use the following table to determine how best to proceed through the book.

If you are	Read through these sections
New to collections	Focus on Parts I and III, or read through the entire book in order.
Interested in .NET built-in collections	Briefly skim Part I if you need a refresher on the core concepts, and then read Part II to learn about the built-in collections Read up on the different technologies that can be used with collections in Part III, and be sure to read Chapter 7 in Part III.
Interested in integrating your collection classes with the .NET Framework	Focus on Parts I and III if you need help creating collection classes and integrating them with the .NET Framework.
Interested in using LINQ with collections	Read through Chapter 7 in Part III.
Interested in using threads with collections	Read through Chapter 8 in Part III.
Interested in serializing your custom collections	Read through Chapter 9 in Part III.
Interested in using collections in your user interface (UI)	Read through the chapters in Part IV.

Most of the book's chapters include hands-on samples that you can use to try out the concepts just explained. No matter which sections you choose to focus on, be sure to download and install the sample applications.

## Conventions and Features in This Book

This book presents information by using conventions designed to make the information readable and easy to follow.

- In most cases, the book includes separate examples for Visual Basic programmers and Visual C# programmers. You can ignore the examples that do not apply to your selected language.
- Boxed elements with labels such as “Note” provide additional information.
- Text that needs to be added to existing code or is being brought to your attention (apart from code blocks) appears in bold.
- A vertical bar between two or more menu items (for example, File | Close) means that you should select the first menu or menu item, then the next, and so on.



**Note** Throughout Chapter 6, you’re asked to add class files to the C# or Visual Basic project you work with in Chapters 1 through 3. You create partial class files the same way you create class files. To create a class file, right-click the project in the Solution Explorer, click Add, and then click Class. From there, type the name of the class or the class file name. If you type the name of the class, Microsoft Visual Studio will create a file by using the name of the class and add a .cs or .vb extension to the file name, depending on whether you are in C# or Visual Basic. If you type a file name, Visual Studio will create a class named the same as the portion of your file name preceding the first period. For example, if you added a class and typed the file name ArrayEx.List.cs or ArrayEx.List.vb, Visual Studio would create a file with that name, but the class within the file would be named ArrayEx (the portion of the name up to the first period).

## System Requirements

You need the following hardware and software to complete the practice exercises in this book:

- One of the following: Windows XP with Service Pack 3 (except Starter Edition), Windows Vista with Service Pack 2 (except Starter Edition), Windows 7, Windows Server 2003 with Service Pack 2, Windows Server 2003 R2, Windows Server 2008 with Service Pack 2, or Windows Server 2008 R2
- Microsoft Visual Studio 2008 or later, any edition (multiple downloads may be required if using Express Edition products)
- The .NET Framework 3.5 or .NET Framework 4.0

- DVD-ROM drive (if installing Visual Studio from DVD)
- Microsoft Silverlight 4 if you want to run any of the Silverlight examples
- An Internet connection to download software or chapter examples

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio.

## Code Samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample projects, in both their pre-exercise and post-exercise formats, can be downloaded from the following page:

<http://go.microsoft.com/fwlink/?LinkId=227007>

Follow the instructions to download the Collections\_sample\_code.zip file.



**Note** In addition to the code samples, your system should have Visual Studio 2008 or Visual Studio 2010 installed. Be sure you have installed the latest service packs for Visual Studio.

## Installing the Code Samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book.

1. Unzip the Collections\_sample\_code.zip file that you downloaded from the book's website (name a specific directory along with directions to create it, if necessary).
2. If prompted, review the displayed end-user license agreement. If you accept the terms, select the accept option, and then click Next.



**Note** If the license agreement doesn't appear, you can access it from the same webpage from which you downloaded the Collections\_sample\_code.zip file.

## Using the Code Samples

The folder structure created by unzipping the download contains a folder for each chapter of the book that has source code and a solution file.

- The source code for each chapter is located under the appropriate chapter number. Each chapter folder contains a folder named VB and one named CS for Visual Basic .NET source code and C# source code respectively.
- All custom collection classes are present in the DevGuideToCollections project in each chapter folder under CS for C# or VB for Visual Basic .NET. DevGuideToCollections is a class library that you can use in your own project.

To view the source code, access the *Developer's Guide to Collections* in the main folder. If your system is configured to display file extensions, the system will display Visual Basic project files with a .vbproj extension and C# project files with .csproj as the file extension.

## Acknowledgments

I would like to thank the team at O'Reilly Media and Microsoft Press for giving me the opportunity to write a book to help fellow developers. I would also like to thank all the developers and companies that have helped me obtain the knowledge to write the contents for this book. I realized how critical it is for developers like myself to have a book like this after hearing and seeing first-hand the issues developers have struggled with in critical applications. Last, but not least, I would like to thank my beautiful wife, friends, neighbors, and coworkers for helping me through the book-writing process; and my son, who joined my wife and me in the middle of writing this book.

## Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at [oreilly.com](http://oreilly.com):

<http://go.microsoft.com/fwlink/?LinkId=227006>

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at

[mspinput@microsoft.com](mailto:mspinput@microsoft.com)

Please note that product support for Microsoft software is not offered through these addresses.

## We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

<http://www.microsoft.com/learning/booksurvey>

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

## Stay in Touch

Let's keep the conversation going! We're on Twitter: <http://twitter.com/MicrosoftPress>.



Part I

# Collection Basics



## Chapter 1

# Understanding Collections: Arrays and Linked Lists

After completing this chapter, you will be able to

- Identify arrays and singly and doubly linked lists.
- Design and implement arrays and singly and doubly linked lists.
- Understand when and when not to use arrays and singly and doubly linked lists.

## Array Overview

An *array* is a collection of elements stored so that you can access each element through an index. Each element is accessed through a simple mathematical formula (*index \* element length*).

## Uses of Arrays

You can use arrays to create other data types. In Chapter 3, “Understanding Collections: Queues, Stacks, and Circular Buffers,” you’ll use an array to create a stack, a queue, and a circular buffer, but you can also use an array to create lists and other collection types, as well as strings and noncollection data types.

Arrays can also represent mathematical concepts such as vectors and matrices. You can create mathematical formulas to perform multiplications, additions, dot product, and so on using arrays that represent matrices and vectors.

You can easily calculate the size and element indices of an array at run time. This makes arrays useful for streaming values to disk or the network and even for passing between dynamic-link libraries (DLLs) created by different programming languages. Libraries are available for quickly copying the data in arrays because the values are stored continuously in memory.



**Note** There is a difference between the contents of an array that contains references and an array that contains values. An array of values is equal to the size of the value type times the number of elements in the array. An array of references is equal to the size of a reference pointer times the number of elements in the array. The size of a reference pointer in a Microsoft .NET Framework 32-bit application is 4 bytes; in a 64-bit application, it's 8 bytes.

Consider using an array when the array contents, but not the properties of the array (such as the size of the array), change frequently. Arrays are very efficient when performing indexing operations, and can quickly access the elements that need to be changed. That speed is mitigated by the overhead required for item removals, insertions, and additions, but some of the overhead can be reduced by implementing different types of arrays. You'll implement an array later in this chapter so that you will have the knowledge you need to implement different types of arrays, depending on your project needs. If you need a general array implementation, you can use the *ArrayList* or *List(T)* class that is discussed in Chapter 4, "Generic Collections."

## Advantages of Arrays

The biggest advantage of arrays is that you can access any item in them using the same mathematical formula instead of having to traverse the collection to find the index you are looking for. Theoretically, it takes the same amount of time to access item 1 as it does to access item 2, 10, 10,000, or 2,300,000. Arrays also require the least amount of memory of all the collection types.

## Disadvantages of Arrays

All items in an array must be laid out in memory so that they can be accessed using a mathematical algorithm. When a user decides to add or remove an item from the array, the new item (or remaining items) must still be accessible using the same mathematical algorithm. This creates a performance penalty for the following reasons:

- When you add an item, you must create a new array and copy the old values to the new array.
- When you remove or insert an item, you must shift up or down all items after the location operated on.



**Note** Some of the performance penalties of adding items to an array can be reduced by allocating more space in the array than it actually requires. That way, you don't have to create a new array to hold the new and old contents until the extra space is used up. This type of array is typically called a *dynamic array*, and it's the type of array you'll implement later in this chapter.

You do not have to truncate an array if you remove an item, but you do have to shift all of the elements that appear after the removed item to fill up the empty space. However, memory will not be freed until you truncate the array. This would not be an issue if you reduced an array from ten items to one item, but it might be an issue if you reduced an array from thousands of items to one item.

# Array Implementation

Now that you have seen the usefulness of arrays, it is time to learn how to implement them. The following sections show you how arrays work and how to implement them.

## Understanding Indexing

An array is formed by sequences of elements that are arranged so that a mathematical algorithm can access each element, as shown in the following illustration.

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

The first element is at index 0, the second at index 1, and the next one is assigned one higher than the previous index until you get to the last index of 8. The indices are from 0 through 8 in the preceding example, which makes this example an array of nine elements. The number of elements can be calculated by adding 1 to the index of the last element because the elements are [0–8] or, written another way, [0]+[1–8].

The preceding example has only nine elements, but an array can contain any number of elements. So how do you refer to the elements in an array that has an unknown number of elements? You say that it has  $N$  elements and refer to each element as shown in the following illustration.

0	1	2	...	$N-1$
---	---	---	-----	-------

The numbers of elements in the list can be calculated by adding 1 to last index. The Count property, shown in the “Adding Helper Methods and Properties” section, returns the number of elements in the array.

To refer to the  $N$  elements’ indices in reverse, you would say the following.

$N-(Count-1)$	$N-(Count-2)$	...	$N-1$	$N-2$
---------------	---------------	-----	-------	-------

## Getting Started

You can create a simple array by using the following code.

C#

```
// Create an array of 13 ints
int[] ints = new int[13];

// Create a string array of colors
string[] colors = { "red", "green", "black" };
```

### Visual Basic

```
' Create an array of 13 ints
Dim ints As New Integer(12) {}

' Create a string array of colors
Dim colors As New String() {"red", "green", "black"}
```

This simple array works fine for simple situations, particularly for static arrays, but it does not help you handle deletions, insertions, notifications, synchronization, removals, or developer security. You will create an array class that you can use instead of a simple array when such issues matter. Microsoft has a class with similar functionality called *ArrayList*. This class is named *ArrayList(T)* to eliminate any possible confusion with the built-in *ArrayList* class. You can find the fully implemented code in the DevGuideToCollections project, in either the Chapter 1\CS\DevGuideToCollections or Chapter 1\VB\DevGuideToCollections folder.



**Note** The *ArrayList* class in the .NET Framework accomplishes the same objectives as this example class. The purpose of this exercise is to show how you can create a custom implementation of the class that fits the needs of your application. The reasons you might want to do this will be discussed later.

First, you need to create a class library to hold the *ArrayList(T)* class so that you can use it in other projects. In Microsoft Visual Studio, create a Microsoft Visual Basic or Microsoft Visual C# class library project. You can name the project *DevGuideToCollections* to make it easier to follow along with the provided samples in the book. After the project is created, create a blank class for the *ArrayList(T)* class and remove the *Class1* class from the project (if the class exists). *ArrayList(T)* is a generic class, so its elements are type-safe. The top-level view of the class is as follows.

### C#

```
namespace DevGuideToCollections
{
    [DebuggerDisplay("Count={Count}")]
    [DebuggerTypeProxy(typeof(ArrayDebugView))]
    public class ArrayEx<T>
    {
        // Fields
        private const int GROW_BY = 10;
        private int m_count;
        private T[] m_data;
        private int m_updateCode;

        // Constructors
        public ArrayEx();
        public ArrayEx(IEnumerable<T> items);
        public ArrayEx(int capacity);
```

```

// Methods
public void Add(T item);
public void Clear();
public bool Contains(T item);
public int IndexOf(T item);
private void Initialize(int capacity);
public void Insert(int index, T item);
public bool Remove(T item);
public bool Remove(T item, bool allOccurrences);
public void RemoveAt(int index);
public T[] ToArray();

// Properties
public int Capacity { get; set; }
public int Count { get; }
public bool IsEmpty { get; }
public T this[int index] { get; set; }
}

}
}

```

**Visual Basic**

```

<DebuggerDisplay("Count={Count}")> _
<DebuggerTypeProxy(typeof(ArrayDebugView))> _
Public Class ArrayEx(Of T)
    ' Fields
    Private Const GROW_BY As Integer = 10
    Private m_count As Integer
    Private m_data As T()
    Private m_updateCode As Integer

    ' Constructors
    Public Sub New()
        Public Sub New(ByVal items As IEnumerable(Of T))
        Public Sub New(ByVal capacity As Integer)

    ' Methods
    Public Sub Add(ByVal item As T)
    Public Sub Clear()
    Public Function Contains(ByVal item As T) As Boolean
    Public Function IndexOf(ByVal item As T) As Integer
    Public Sub Initialize(ByVal capacity As Integer)
    Public Sub Insert(ByVal index As Integer, ByVal item As T)
    Public Function Remove(ByVal item As T) As Boolean
    Public Function Remove(ByVal item As T, ByVal allOccurrences As Boolean) As Boolean
    Public Sub RemoveAt(ByVal index As Integer)
    Public Function ToArray() As T()

    ' Properties
    Public Property Capacity As Integer
    Public ReadOnly Property Count As Integer
    Public ReadOnly Property IsEmpty As Boolean
    Public Default Property Item(ByVal index As Integer) As T

End Class

```

All elements will be stored in the simple array *m\_data*. The *m\_count* field will track how many of the elements in *m\_data* are actually being used. The *GROW\_BY* constant will be explained later in this section.

The *m\_updateCode* field will be incremented each time the user modifies the list. You'll use the *m\_updateCode* field in Chapter 6, ".NET Collection Interfaces," to determine if the collection has changed while the user is iterating over it. It is easier to add it to the code now rather than change the code in Chapter 6.



**Note** The *DebuggerDisplayAttribute* is used to control how the custom collection classes are displayed in the debugger. The specified parameters tell the debugger to use "Count=[{Count}]" instead of the *ToString* method. The *DebuggerTypeProxyAttribute* is used to specify a proxy for the object being displayed in the debugger. The *ArrayDebugView* class returns items stored in the collection to the debugger. The source code for the proxy is located in the Chapter 1 \CS\DevGuideToCollections\ArrayDebugView.cs file for C# and in the Chapter 1\VB \DevGuideToCollections\ArrayDebugView.vb file for Visual Basic.

## Creating Constructors

The *ArrayEx(T)* class will contain three constructors. One constructor is for creating an empty class, one is for creating a class with an initial capacity, and the other is for creating a class with default values. Rather than implement the same functionality in all three constructors, you use a method named *Initialize* at the beginning of all three constructors.

C#

```
void Initialize(int capacity)
{
    m_data = new T[capacity];
}
```

Visual Basic

```
Sub Initialize(ByVal capacity As Integer)
    m_data = New T(capacity - 1) {}
End Sub
```



**Note** In the statement, *New T(X)*, Visual Basic creates an array of *T*s from 0 through *X*, because it considers the *X* to be an upper bound instead of a count. To correct this, you must state *(X-1)* instead.

The *Initialize* method creates a simple array of the size passed in. You could add to this method other initializations that are common among all constructors as well.

Next, create a default constructor for the *ArrayEx(T)* class so that the user can create an empty array.

**C#**

```
/// <summary>
/// Initializes a new instance of the ArrayEx(T) class that is empty./// </summary>
public ArrayEx()
{
    Initialize(GROW_BY);
}
```

**Visual Basic**

```
''' <summary>
''' Initializes a new instance of the ArrayEx(T) class that is empty.
''' </summary>
Public Sub New()
    Initialize(GROW_BY)
End Sub
```

The default constructor creates an internal array of size *GROW\_BY* and a *count* of 0. In the “Disadvantages of Arrays” section, you saw how adding items can have a performance penalty. This performance penalty can be reduced by adding what is called a *grow-by* value. Instead of increasing the internal array by one each time you need to add a value, you increase the array by the *GROW\_BY* size. That way, you won’t have to increase the size again until all the elements added by the grow-by are used up. Bear in mind that there is a trade-off: If the grow-by size is too large, you use memory unnecessarily; if it’s too small, you will still spend a lot of time reallocating the array. You can eliminate some of these problems by introducing a *capacity* property, which would allow the user to change the capacity before adding multiple items. You can allow the user to create an array with an initial capacity by using the following constructor. You will learn more about *capacity* later in this section.

**C#**

```
/// <summary>
/// Initializes a new instance of the ArrayEx(T) class that is empty and has the
/// specified initial capacity.
/// </summary>
/// <param name="capacity">
/// The number of elements that the new array can initially store.
/// </param>
public ArrayEx(int capacity)
{
    Initialize(capacity);
}
```

**Visual Basic**

```
''' <summary>
''' Initializes a new instance of the ArrayEx(T) class that is empty and has the
''' specified initial capacity.
''' </summary>
''' <param name="capacity">
''' The number of elements that the new array can initially store.
''' </param>
Public Sub New(ByVal capacity As Integer)
    Initialize(capacity)
End Sub
```

With this constructor, you can create an array with an initial internal size. If you know you will be adding a large number of items to the array, you can create one with the size you need.

Creating an array that already contains items is also useful. You can use the following constructor to do so.

### C#

```
/// <summary>
/// Initializes a new instance of the ArrayEx(T) class that contains the items in the array.
/// </summary>
/// <param name="items">Adds the items to the ArrayEx(T).</param>
public ArrayEx(IEnumerable<T> items)
{
    Initialize(GROW_BY);

    foreach (T item in items)
    {
        Add(item);
    }
}
```

### Visual Basic

```
''' <summary>
''' Initializes a new instance of the ArrayEx(T) class that contains the items in the array.
''' </summary>
''' <param name="items">Adds the items to the end of the list.</param>
Public Sub New(ByVal items As IEnumerable(Of T))
    Initialize(GROW_BY)

    For Each item As T In items
        Add(item)
    Next
End Sub
```

Each item in *items* will be added to the end of the array in the order it was passed in.

## Allowing Users to Add Items

The array would be useless if you could never add anything to it. To add items, you need to add methods for both adding and inserting items into the array. This can be accomplished through the *Add* and *Remove* methods.

You can use the following method to add items to the end of the array.

### C#

```
/// <summary>
/// Adds an object to the end of the ArrayEx(T).
/// </summary>
/// <param name="item">The item to add to the end of the ArrayEx(T).</param>
```

```
public void Add(T item)
{
    if (m_data.Length <= m_count)
    {
        Capacity += GROW_BY;
    }

    // We will need to assign the item to the last element and then increment
    // the count variable
    m_data[m_count++] = item;
    ++m_updateCode;
}
```

### Visual Basic

```
''' <summary>
''' Adds an object to the end of the ArrayEx(T).
''' </summary>
''' <param name="item">The item to add to the end of the ArrayEx(T).</param>
Public Sub Add(ByVal item As T)
    If (m_data.Length <= m_count) Then
        Capacity += GROW_BY
    End If

    ' We will need to assign the item to the last element and then increment
    ' the count variable
    m_data(m_count) = item
    m_count += 1
    m_updateCode += 1
End Sub
```

When the method is called, it will add an item to the end of the array and increment *m\_count*. If there is no room in *m\_data* for the item, it will increment the capacity of the *m\_data* by *GROW\_BY* as explained earlier in this chapter.

Sometimes you need to add an item to the array in a position other than the end. You can use the following insertion method to insert an item into the array at any position.

### C#

```
/// <summary>
/// Inserts an item into the ArrayEx(T) at the specified index.
/// </summary>
/// <param name="index">The zero-based index at which item should be inserted.</param>
/// <param name="item">
/// The item to insert. A value of null will cause an exception later.
/// </param>

public void Insert(int index, T item)
{
    if (index < 0 || index >= m_count)
    {
        throw new ArgumentOutOfRangeException("index");
    }
}
```

```

    if (m_count + 1 >= Capacity)
    {
        Capacity = m_count + GROW_BY;
    }

    // First we need to shift all elements at the location up by one
    for (int i = m_count; i > index && i > 0; --i)
    {
        m_data[i] = m_data[i - 1];
    }

    m_data[index] = item;

    ++m_count;
    ++m_updateCode;
}

```

### Visual Basic

```

''' <summary>
''' Inserts an item into the ArrayEx(T) at the specified index.
''' </summary>
''' <param name="index">The zero-based index at which item should be inserted.</param>
''' <param name="item">
''' The item to insert. A value of Nothing will cause an exception later.
''' </param>
Public Sub Insert(ByVal index As Integer, ByVal item As T)

    If (index < 0 Or index >= m_count) Then
        Throw New ArgumentOutOfRangeException("index")
    End If

    If (m_count + 1 >= Capacity) Then
        Capacity = m_count + GROW_BY
    End If

    ' First we need to shift all elements at the location up by one
    Dim i As Integer = m_count
    While (i > index And i > 0)
        m_data(i) = m_data(i - 1)
        i -= 1
    End While

    m_data(index) = item

    m_count += 1
    m_updateCode += 1
End Sub

```

When you insert an item, the item located at the insertion point and all items after need to be shifted up one index position to make room for the new item. Note that you perform that shift starting with the last item in the array, to avoid overwriting any items. As with the *Add* method, this method adjusts the capacity of the array to hold the item.

## Allowing Users to Remove Items

Users need to be able to remove items that they no longer need from the *ArrayEx(T)* classes. With the *Clear*, *RemoveAt*, and *Remove* methods, users can do so. Because changing the capacity of the array is very costly, none of the methods change the capacity of the array. You need to implement a method to do so if you choose.

If you have only the index of the item you need to remove, you can use the following method to remove an item by its index only.

### C#

```
/// <summary>
/// Removes the item located at the specified index.
/// </summary>
/// <param name="index">The index of the item to remove</param>
public void RemoveAt(int index)
{
    if (index < 0 || index >= m_count)
    {
        // Item has already been removed.
        return;
    }

    int count = Count;

    // Shift all of the elements after the specified index down one.
    for (int i = index + 1; i < count; ++i)
    {
        m_data[i - 1] = m_data[i];
    }

    // Decrement the count to reflect the item being removed.
    --m_count;
    ++m_updateCode;

    m_data[m_count] = default(T);
}
```

### Visual Basic

```
''' <summary>
''' Removes the item located at the specified index.
''' </summary>
''' <param name="index">The index of the item to remove</param>
Public Sub RemoveAt(ByVal index As Integer)

    If (index < 0 Or index >= m_count) Then
        ' Item has already been removed.
        Return
    End If
```

```

Dim count As Integer = Me.Count

' Shift all of the elements after the specified index down one.
For i As Integer = index + 1 To count - 1
    m_data(i - 1) = m_data(i)
Next

' Decrement the count to reflect the item being removed.
m_count -= 1
m_updateCode += 1

m_data(m_count) = CType(Nothing,T)
End Sub

```

The *RemoveAt* method removes the item located at the specified index and adjusts the count to reflect the change. All items after the item are shifted down one index to fill the slot created from the removed item. This leaves the last slot empty, which is then set to the default value. Setting a slot to the default value removes the reference to the object instance so that it can be garbage collected.

Removing by the index is the fastest approach, but the index is not always known. For those cases, you need to use the following methods.

### C#

```

/// <summary>
/// Removes the first occurrence of the specified item from the ArrayEx(T).
/// </summary>
/// <param name="item">The item to remove from the ArrayEx(T).</param>
/// <returns>True if an item was removed, false otherwise.</returns>
public bool Remove(T item)
{
    return Remove(item, false);
}

/// <summary>
/// Removes the first or all occurrences of the specified item from the ArrayEx(T).
/// </summary>
/// <param name="item">The item to remove from the ArrayEx(T).</param>
/// <param name="allOccurrences">
/// True if all occurrences of the item should be removed, False if only the first
/// should be removed.
/// </param>
/// <returns>True if an item was removed, false otherwise.</returns>
public bool Remove(T item, bool allOccurrences)
{
    int shiftto = 0;
    bool shiftmode = false;
    bool removed = false;

    int count = m_count;
    EqualityComparer<T> comparer = EqualityComparer<T>.Default;

    for (int i = 0; i < count; ++i)

```

```

{

    if (comparer.Equals(m_data[i], item) && (allOccurrences || !shiftmode))
    {
        // Decrement the count since we have found an instance
        --m_count;
        removed = true;

        // Check to see if we have already found one occurrence of the
        // item we are removing
        if (!shiftmode)
        {
            // We will start shifting to the position of the first occurrence.
            shiftto = i;
            // Enable shifting
            shiftmode = true;
        }

        continue;
    }

    if (shiftmode)
    {
        // Since we are shifting elements we need to shift the element
        // down and then update the shiftto index to the next element.
        m_data[shiftto++] = m_data[i];
    }
}

for (int i = m_count; i < count; ++i)
{
    m_data[i] = default(T);
}

if (removed)
{
    ++m_updateCode;
}

return removed;
}

```

### Visual Basic

```

''' <summary>
''' Removes the first occurrence of the specified item from the ArrayEx(T).
''' </summary>
''' <param name="item">The item to remove from the ArrayEx(T).</param>
''' <returns>True if an item was removed, false otherwise.</returns>
Public Function Remove(ByVal item As T) As Boolean
    Return Remove(item, False)
End Function

''' <summary>
''' Removes the first or all occurrences of the specified item from the ArrayEx(T).
''' </summary>

```

```

''' <param name="item">The item to remove from the ArrayEx(T).</param>
''' <param name="allOccurrences">
''' True if all occurrences of the item should be removed,
''' False if only the first should be removed.
''' </param>
''' <returns>True if an item was removed, false otherwise.</returns>
Public Function Remove(ByVal item As T, ByVal allOccurrences As Boolean) As Boolean
    Dim shiftto As Integer = 0
    Dim shiftmode As Boolean = False
    Dim removed As Boolean = False

    Dim count As Integer = m_count
    Dim comparer As EqualityComparer(Of T) = EqualityComparer(Of T).Default

    For i As Integer = 0 To count - 1

        If (comparer.Equals(m_data(i), item) And (allOccurrences Or Not shiftmode)) Then
            ' Decrement the count since we have found an instance
            m_count -= 1
            removed = True

            ' Check to see if we have already found one occurrence of the
            ' item we are removing
            If (Not shiftmode) Then
                ' We will start shifting to the position of the first occurrence.
                shiftto = i
                ' Enable shifting
                shiftmode = True
            End If

            Continue For
        End If

        If (shiftmode) Then

            ' Since we are shifting elements we need to shift the element
            ' down and then update the shiftto index to the next element.
            m_data(shiftto) = m_data(i)
            shiftto += 1
        End If

    Next

    Return removed
End Function

```

The *Remove* method linearly searches the list for the item that needs to be removed. After it removes the item, it shifts the items after it down one index to fill the slot created by the removed item. The *allOccurrences* flag lets the user remove all occurrences of the item without having to call *Remove* repeatedly until a *false* is returned.

Rather than calling the *RemoveAt* method repeatedly until the count is 0 to remove all items from the array, you can take a simpler approach by using the following method to accomplish that task.

#### C#

```
/// <summary>
/// Clears all values from the ArrayEx(T).
/// </summary>
public void Clear()
{
    Array.Clear(m_data,0,m_count);
    m_count = 0;
    ++m_updateCode;
}
```

#### Visual Basic

```
''' <summary>
''' Clears all values from the ArrayEx(T).
''' </summary>
Public Sub Clear()
    Array.Clear(m_data,0,m_count)
    m_count = 0
    m_updateCode += 1
End Sub
```

The *Clear* method changes the count to 0. The *Array.Clear* method sets all values to the default value so that garbage collection can remove any items that are no longer being referenced.

## Adding Helper Methods and Properties

Users will want the ability to check the status of the array. The *Contains* and *IndexOf* methods and the *Item* property allow users to look at the contents of the array, whereas the *Capacity*, *Count*, and *IsEmpty* properties allow them to look at the status of the array.

Your users may find it necessary to get the index of an item in the array or simply check to see if an item is in the array. This information can stop them from having to unnecessarily traverse or operate on the array. The following methods allow them to do so.

#### C#

```
/// <summary>
/// Checks to see if the item is present in the ArrayEx(T).
/// </summary>
```

```

/// <param name="item">The item to see if the array contains.</param>
/// <returns>True if the item is in the array, false if it is not.</returns>
public bool Contains(T item)
{
    EqualityComparer<T> comparer = EqualityComparer<T>.Default;
    for (int i = 0; i < m_count; i++)
    {
        if (comparer.Equals(m_data[i], item))
        {
            return true;
        }
    }
    return false;
}

/// <summary>
/// Gets the index of the specified item.
/// </summary>
/// <param name="item">The item to get the index of.</param>
/// <returns>
/// -1 if the item isn't found in the array, the index of the first instance of the
/// item otherwise.
/// </returns>
public int IndexOf(T item)
{
    return Array.IndexOf<T>(m_data, item, 0, m_count);
}

```

### Visual Basic

```

''' <summary>
''' Checks to see if the item is present in the ArrayEx(T).
''' </summary>
''' <param name="item">The item to see if the array contains.</param>
''' <returns>True if the item is in the array, false if it is not.</returns>
Public Function Contains(ByVal item As T) As Boolean
    Dim comparer As EqualityComparer(Of T) = EqualityComparer(Of T).Default
    For i As Integer = 0 To m_count - 1
        If (comparer.Equals(m_data(i), item)) Then
            Return True
        End If
    Next
    Return False
End Function

''' <summary>
''' Gets the index of the specified item.
''' </summary>
''' <param name="item">The item to get the index of.</param>
''' <returns>
''' -1 if the item isn't found in the array, the index of the found item otherwise.
''' </returns>
Public Function IndexOf(ByVal item As T) As Integer
    Return Array.IndexOf(Of T)(m_data, item, 0, m_count)
End Function

```

The *Contains* method allows users to see whether an item is in the array, and the *IndexOf* method allows them to get an index of the item they specified. The *Contains* method uses the equality comparer to find the specified item in the array. You learn more about the equality comparer in Chapter 4. The *IndexOf* method uses the *Array.IndexOf* method to find the index of the item in the array.

But what if your users want to check the contents at an index or even change its value? The following property allows just that.

### C#

```
/// <summary>
/// Gets or sets an element in the ArrayEx(T).
/// </summary>
/// <param name="index">The index of the element.</param>
/// <returns>The value of the element.</returns>
public T this[int index]
{
    get
    {
        if (index < 0 || index >= m_count)
        {
            throw new ArgumentOutOfRangeException("index");
        }

        return m_data[index];
    }
    set
    {
        if (index < 0 || index >= m_count)
        {
            throw new ArgumentOutOfRangeException("index");
        }

        m_data[index] = value;
        ++m_updateCode;
    }
}
```

### Visual Basic

```
''' <summary>
''' Gets or sets an element in the ArrayEx(T).
''' </summary>
''' <param name="index">The index of the element.</param>
''' <returns>The value of the element.</returns>
Default Public Property Item(ByVal index As Integer) As T
    Get
        If (index < 0 Or index >= m_count) Then
            Throw New ArgumentOutOfRangeException("index")
        End If

        Return m_data(index)
    End Get
```

```

Set(ByVal value As T)
    If (index < 0 Or index >= m_count) Then
        Throw New ArgumentException("index")
    End If

    m_data(index) = value
    m_updateCode += 1
End Set
End Property

```

The *Item* property allows the user to directly set or get the contents at a specific index. The array index value is any value that satisfies the condition  $0 \leq index < Count$ . However, without a *Count* property, the user would never know the bounds of the array. The following properties will help you to know when to do an operation and the range of the array.



**Note** C# uses the *this* keyword to implement the *Item* property. Visual Basic actually implements the *Item* property. Both implement the same indexing functionality.

### C#

```

/// <summary>
/// Gets the number of elements actually contained in the ArrayEx(T).
/// </summary>
public int Count
{
    get { return m_count; }
}

/// <summary>
/// States if the ArrayEx(T) is empty.
/// </summary>
public bool IsEmpty
{
    get { return m_count <= 0; }
}

```

### Visual Basic

```

''' <summary>
''' Gets the number of elements actually contained in the ArrayEx(T).
''' </summary>
Public ReadOnly Property Count() As Integer
    Get
        Return m_count
    End Get
End Property

''' <summary>
''' States if the ArrayEx(T) is empty.
''' </summary>

```

```
Public ReadOnly Property IsEmpty() As Boolean
    Get
        Return m_count <= 0
    End Get
End Property
```

The *Count* property returns the number of items in the array, and the *IsEmpty* property states whether there are any items in the array.

As you saw earlier in the chapter, the capacity of the array can be used to help with performance issues. A user can use the capacity to eliminate constant resizing when doing multiple adds by changing the capacity of the array. The following property allows the user to change the capacity.

### C#

```
/// <summary>
/// Gets or sets the size of the internal data array.
/// </summary>
public int Capacity
{
    get { return m_data.Length; }
    set
    {
        // We do not support truncating the stored array.
        // So throw an exception if the array is less than Count.
        if (value < Count)
        {
            throw new ArgumentOutOfRangeException("value", "The value is less than Count");
        }

        // We do not need to do anything if the newly specified capacity
        // is the same as the old one.
        if (value == Capacity)
        {
            return;
        }

        // We will need to create a new array and move all of the
        // values in the old array to the new one
        T[] tmp = new T[value];

        for (int i = 0; i < Count; ++i)
        {
            tmp[i] = m_data[i];
        }

        m_data = tmp;
        ++m_updateCode;
    }
}
```

**Visual Basic**

```

''' <summary>
''' Gets or sets the size of the internal data array.
''' </summary>
Public Property Capacity() As Integer
    Get
        Return m_data.Length
    End Get
    Set(ByVal value As Integer)
        ' We do not support truncating the stored array.
        ' So throw an exception if the array is less than Count.
        If (value < m_count) Then
            Throw New ArgumentOutOfRangeException("value", "The value is less than Count")
        End If

        ' We do not need to do anything if the newly specified capacity
        ' is the same as the old one.
        If (value = Capacity) Then
            Exit Property
        End If

        ' We will need to create a new array and move all of the values
        ' in the old array to the new one
        Dim tmp As T() = New T(value - 1) {}

        For i As Integer = 0 To m_count - 1
            tmp(i) = m_data(i)
        Next

        m_data = tmp
        m_updateCode += 1
    End Set
End Property

```

Finally, the following helper function copies the contents of the internal array into a new array. The new array will be useful when code requires a basic array rather than an instance of the *ArrayEx(T)* class.

**C#**

```

/// <summary>
/// Copies the elements of the ArrayEx<T> to a new array.
/// </summary>
/// <returns>An array containing copies of the elements of the ArrayEx<T>. </returns>
public T[] ToArray()
{
    T[] tmp = new T[Count];

    for (int i = 0; i < Count; ++i)
    {
        tmp[i] = m_data[i];
    }

    return tmp;
}

```

**Visual Basic**

```
''' <summary>
''' Copies the elements of the ArrayEx(T) to a new array.
''' </summary>
''' <returns>An array containing copies of the elements of the ArrayEx(T).</returns>
Public Function ToArray() As T()
    Dim tmp As New T(m_count - 1) {}

    For i As Integer = 0 To m_count - 1
        tmp(i) = m_data(i)
    Next

    Return tmp
End Function
```

## Using the *ArrayEx(T)* Class

Suppose that you have just been given a mission by your boss at 5:59 P.M. Your mission, if you choose to accept it, is to write a console application that adds 20 random numbers to an array and then sorts the array. Because you were supposed to leave a long time ago, you decide to use a simple selection sort for the sorting and the *ArrayEx(T)* class you just created.



**Note** You can see the finished example for this section, and all sections in this chapter, in the Chapter 1\CS\Driver folder for C# or the Chapter 1\VB\Driver folder for Visual Basic.

First, create a C# or Visual Basic console application named *Driver* in Visual Studio. Right-click the solution in the Solution Explorer, and select Add | Existing Project. Browse for the DevGuideToCollections project that you just created, and then click Add. Right-click the Driver project in the Solution Explorer, and select Add Reference. Click the Projects tab, and select the DevGuideToCollections project. Next, open Program.cs in C# or Module1.vb in Visual Basic in the Driver project by using the Solution Explorer. At the top of the file, add the following lines.

**C#**

```
using DevGuideToCollections;
using System.Text;
```

**Visual Basic**

```
Imports DevGuideToCollections
Imports System.Text
```

For C#, in the class *Program*, create a method called *Lesson1a* as follows.

**C#**

```
static void Lesson1A()
{
}
```

For Visual Basic, in the module *Module1*, create a method called *Lesson1a* as follows.

#### Visual Basic

```
Sub Lesson1A()
End Sub
```

Create the following helper method.

#### C#

```
static string ArrayToString(Array array)
{
    StringBuilder sb = new StringBuilder();

    sb.Append("[");
    if (array.Length > 0)
    {
        sb.Append(array.GetValue(0));
    }
    for (int i = 1; i < array.Length; ++i)
    {
        sb.AppendFormat(",{0}", array.GetValue(i));
    }
    sb.Append("]");

    return sb.ToString();
}
```

#### Visual Basic

```
Function ArrayToString(ByVal array As Array) As String
    Dim sb As New StringBuilder()

    sb.Append("[")
    If (array.Length > 0) Then
        sb.Append(array.GetValue(0))
    End If
    For i As Integer = 1 To array.Length - 1
        sb.AppendFormat(",{0}", array.GetValue(i))
    Next
    sb.Append("]")

    Return sb.ToString()
End Function
```

The helper method converts the elements of an array to a string.



**More Info** In Chapter 6, you see how to traverse the *ArrayEx(T)* with a *foreach* statement.

Add the following lines of code to the method *Lesson1a*. These lines of code will create a random array of integers from 0 through 100.

**C#**

```
Random rnd = new Random();
ArrayEx<int> array = new ArrayEx<int>();

for (int i = 0; i < 20; ++i)
{
    array.Add(rnd.Next(100));
}
```

**Visual Basic**

```
Dim rnd As Random = New Random()
Dim array As ArrayEx(Of Integer) = New ArrayEx(Of Integer)()

For i As Integer = 0 To 19
    array.Add(rnd.Next(100))
Next
```

*Random rnd = new Random()* creates an instance of the class *Random*, which is used to create random numbers. The *rnd.Next(100)* call returns a random integer from 0 through 100.

The following lines of code will write the unsorted list to the console.

**C#**

```
Console.WriteLine("Sorting the following list");
Console.WriteLine(ArrayToString(array.ToArray()));
```

**Visual Basic**

```
Console.WriteLine("Sorting the following list")
Console.WriteLine(ArrayToString(array.ToArray()))
```

The following lines of code will perform a selection sort on the array that you created.

**C#**

```
for (int i = 0; i < array.Count; ++i)
{
    for (int j = i + 1; j < array.Count; ++j)
    {
        if (array[i] > array[j])
        {
            int tmp = array[j];
            array[j] = array[i];
            array[i] = tmp;
        }
    }
}
```

**Visual Basic**

```

For i As Integer = 0 To array.Count - 1
    For j As Integer = i + 1 To array.Count - 1
        If (array(i) > array(j)) Then
            Dim tmp As Integer = array(j)
            array(j) = array(i)
            array(i) = tmp
        End If
    Next
Next

```

The following lines of code will write the sorted list to the console.

**C#**

```

Console.WriteLine("The sorted list is");
Console.WriteLine(ArrayToString(array.ToArray()));

```

**Visual Basic**

```

Console.WriteLine("The sorted list is")
Console.WriteLine(ArrayToString(array.ToArray()))

```

Scroll down to the *Main* method and add the following to the method.

**C#**

```

Lesson1A();
Console.WriteLine("press enter to continue");
Console.ReadLine();

```

**Visual Basic**

```

Lesson1A()
Console.WriteLine("press enter to continue")
Console.ReadLine()

```

With the *Console.ReadLine*, you can see the console output before being returned to Visual Studio when in debug mode. You need to press Enter on the keyboard to exit the console application.

The following is an example of what is displayed in the console.

**Output**

```

Sorting the following list
[17,17,54,65,74,54,9,58,36,54,30,59,21,31,81,0,71,68,23,75,87]
The sorted list is
[0,0,9,17,21,23,30,31,36,54,54,54,58,59,65,68,71,74,75,81,87]
press enter to continue

```

## Linked List Overview

A *linked list* is a collection of nodes, each of which references the next and/or previous node in a sequence. Items in the list do not have to be unique, and they do not have to be placed into the list in a sequence order; they can be added to the list in any order. Linked lists are normally accessed through a node mechanism, whereas arrays are normally accessed through indexes.

## Uses of Linked Lists

Like arrays, linked lists are typically used to implement other data structures. You'll use a linked list in Chapter 3 to implement a queue and a stack, but linked lists can be used to create other collection types as well.

Consider using a linked list if you plan to do a lot of insertions, removals, or additions. Linked lists are not very efficient at indexing operations, but they perform pretty well when traversing the list.



**Note** Always remember to weigh the advantages of indexing over inserting, removing, or adding elements. A linked list may not be your best choice if you are constantly indexing and adding to a collection of thousands of elements. An array with a large grow size may be your best bet in that case. However, if you are constantly inserting and removing from the collection, a link list may be your best choice. Keep in mind that walking from the beginning of a linked list to the end can be almost (or as) efficient as doing the same with an array, depending on the implementation of the linked list. Also, you can use the previous and next pointers of the linked list node if you are traversing the sequence from the previous access node.

## Advantages of Linked Lists

Because each node in the list references the next node in the sequence, you do not get the shifting and reallocating performance hit that occurs with arrays. Deletions and insertions can be even faster if you have the node that you are removing or inserting after. Nodes can be easily rearranged by only changing their references. Doubly linked lists have an additional reference that makes adds and some removals more efficient than in a singly linked list.

## Disadvantages of Linked Lists

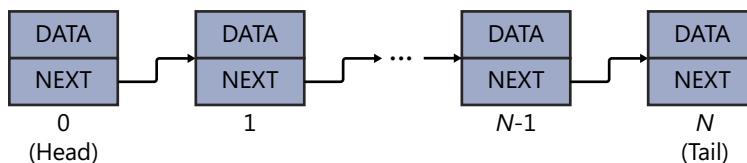
Lists require more memory than arrays because each node references the next node. Also, items are not arranged so that a mathematical formula can be used to access each element, so elements have to be traversed in order to find the item located at the index you want. You may have to traverse the list during a removal and insertion if you have the data and not the node of what you want to operate on. Doubly linked lists also require more memory than singly linked lists.

## Linked List Implementation

Now that you have seen the usefulness of linked lists, it is time to implement them. The following sections show you how linked lists work and how to implement them.

### Singly Linked List Implementation

In a singly linked list, each node contains data and a reference to the next node in a sequence. You can traverse forward through the list from any node but you need to start from the beginning to find the node that precedes the current node in the sequence.



**Note** There is no rule that states you must create your node structure like the one shown in the preceding illustration. You could have the data contain the node information instead. This may be useful if you are implementing custom objects that are passed around in your application and need to be constantly removed and inserted. It may save you the performance hit of having to traverse the collection to find the node. It may also be helpful if several values in the list equal each other but you want to interact with the exact one you are handed.

### Creating the Node Class

The first node is called the *head* and the last node is called *tail*. Each node is represented by a class that is called *SingleLinkedListNode(T)* and is defined as follows.

C#

```

/// <summary>
/// Represents a node in a SingleLinkedList(T).
/// </summary>
/// <typeparam name="T">Specifies the type of data in the node.</typeparam>
  
```

```
[DebuggerDisplay("Data={Data}")]
public class SingleLinkedListNode<T>
{
    SingleLinkedList<T> m_owner;
    SingleLinkedListNode<T> m_next;
    T m_data;

    /// <summary>
    /// Initializes a new instance of the SingleLinkedList(T) class with the specified data.
    /// </summary>
    /// <param name="data">The data that this node will contain.</param>
    public SingleLinkedListNode(T data)
    {
        m_data = data;
    }

    /// <summary>
    /// Initializes a new instance of the SingleLinkedList(T) class
    /// with the specified data and owner.
    /// </summary>
    /// <param name="data">The data that this node will contain.</param>
    internal SingleLinkedListNode(SingleLinkedList<T> owner, T data)
    {
        m_data = data;
        m_owner = owner;
    }

    /// <summary>
    /// Returns the next node.
    /// </summary>
    public SingleLinkedListNode<T> Next
    {
        get { return m_next; }
        internal set { m_next = value; }
    }

    /// <summary>
    /// Gets or sets the owner of the node.
    /// </summary>
    internal SingleLinkedList<T> Owner
    {
        get { return m_owner; }
        set { m_owner = value; }
    }

    /// <summary>
    /// Gets the data contained in the node.
    /// </summary>
    public T Data
    {
        get { return m_data; }
        internal set { m_data = value; }
    }
}
```

**Visual Basic**

```
''' <summary>
''' Represents a node in a SingleLinkedList(T).
''' </summary>
''' <typeparam name="T">Specifies the type of data in the node.</typeparam>
<DebuggerDisplay("Data={Data}")> _
Public Class SingleLinkedListNode(Of T)
    Private m_owner As SingleLinkedList(Of T)
    Private m_next As SingleLinkedListNode(Of T)
    Private m_data As T

    ''' <summary>
    ''' Initializes a new instance of the SingleLinkedList(T) class with the specified data.
    ''' </summary>
    ''' <param name="data">The data that this node will contain.</param>
    Public Sub New(ByVal data As T)
        m_data = data
        m_owner = Nothing
    End Sub

    ''' <summary>
    ''' Initializes a new instance of the SingleLinkedList(T) class
    ''' with the specified data and owner.
    ''' </summary>
    ''' <param name="data">The data that this node will contain.</param>
    Friend Sub New(ByVal owner As SingleLinkedList(Of T), ByVal data As T)
        m_data = data
        m_owner = owner
    End Sub

    ''' <summary>
    ''' Returns the next node.
    ''' </summary>
    Public Property [Next]() As SingleLinkedListNode(Of T)
        Get
            Return m_next
        End Get
        Friend Set(ByVal value As SingleLinkedListNode(Of T))
            m_next = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets the owner of the node.
    ''' </summary>
    Friend Property Owner() As SingleLinkedList(Of T)
        Get
            Return m_owner
        End Get
        Set(ByVal value As SingleLinkedList(Of T))
            m_owner = value
        End Set
    End Property
```

```
''' <summary>
''' Gets the data contained in the node.
''' </summary>
Public Property Data() As T
    Get
        Return m_data
    End Get
    Set(ByVal value As T)
        m_data = value
    End Set
End Property

End Class
```

The class is defined as a generic class, so the contained data will remain type-safe. The field *m\_data* contains the data that you want to store. The field *m\_next* references the next node in the list. The property that wraps *m\_next* is internal so that users can't change the "next" pointer and bypass the bookkeeping. The field *m\_owner* is used to verify that the node belongs to the list that is being operated on.

## Declaring the *SingleLinkedList(T)* Class

The *SingleLinkedList(T)* class is implemented as a singly linked list and is defined as follows.

C#

```
[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(ArrayDebugView))]
public class SingleLinkedList<T>
{
    // Fields
    private int m_count;
    private SingleLinkedListNode<T> m_head;
    private SingleLinkedListNode<T> m_tail;
    private int m_updateCode;

    // Constructors
    public SingleLinkedList();
    public SingleLinkedList(IEnumerable<T> items);

    // Methods
    public SingleLinkedListNode<T> AddAfter(SingleLinkedListNode<T> node, T value);
    public void AddAfter(SingleLinkedListNode<T> node, SingleLinkedListNode<T> newNode);
    public SingleLinkedListNode<T> AddBefore(SingleLinkedListNode<T> node, T value);
    public void AddBefore(SingleLinkedListNode<T> node, SingleLinkedListNode<T> newNode);
    public SingleLinkedListNode<T> AddToBeginning(T value);
    public SingleLinkedListNode<T> AddToEnd(T value);
    public void Clear();
    public bool Contains(T data);
    public SingleLinkedListNode<T> Find(T data);
    public bool Remove(T item);
    public void Remove(SingleLinkedListNode<T> node);
    public bool Remove(T item, bool allOccurrences);
    public T[] ToArray();
```

```
// Properties
public int Count { get; }
public SingleLinkedListNode<T> Head { get; private set; }
public bool IsEmpty { get; }
public SingleLinkedListNode<T> Tail { get; private set; }
}
```

### Visual Basic

```
Public Class SingleLinkedList(Of T)
    ' Constructors
    Public Sub New()
    Public Sub New(ByVal items As IEnumerable(Of T))

    ' Methods
    Public Function AddAfter(ByVal node As SingleLinkedListNode(Of T), ByVal value As T) _
        As SingleLinkedListNode(Of T)
    Public Sub AddAfter(ByVal node As SingleLinkedListNode(Of T), ByVal newNode _ 
        As SingleLinkedListNode(Of T))
    Public Function AddBefore(ByVal node As SingleLinkedListNode(Of T), ByVal value As T) _ 
        As SingleLinkedListNode(Of T)
    Public Sub AddBefore(ByVal node As SingleLinkedListNode(Of T), ByVal newNode _ 
        As SingleLinkedListNode(Of T))
    Public Function AddToBeginning(ByVal value As T) As SingleLinkedListNode(Of T)
    Public Function AddToEnd(ByVal value As T) As SingleLinkedListNode(Of T)
    Public Sub Clear()
    Public Function Contains(ByVal data As T) As Boolean
    Public Function Find(ByVal data As T) As SingleLinkedListNode(Of T)
    Public Function Remove(ByVal item As T) As Boolean
    Public Sub Remove(ByVal node As SingleLinkedListNode(Of T))
    Public Function Remove(ByVal item As T, ByVal allOccurrences As Boolean) As Boolean
    Public Function ToArray() As T()

    ' Properties
    Public ReadOnly Property Count As Integer
    Property Head As SingleLinkedListNode(Of T)
    Public ReadOnly Property IsEmpty As Boolean
    Property Tail As SingleLinkedListNode(Of T)

    ' Fields
    Private m_count As Integer
    Private m_head As SingleLinkedListNode(Of T)
    Private m_tail As SingleLinkedListNode(Of T)
    Private m_updateCode As Integer
End Class
```

For performance reasons, *SingleLinkedList(T)* maintains a reference to the last node in the field *m\_tail*. The *m\_tail* field provides quick access to the last item when a user wants to add a node to the list. This eliminates the need to traverse the list to find the last node. The field *m\_head* references the first node, and the field *m\_count* tracks the number of nodes in the list.

The *m\_updateCode* field will be incremented each time the user modifies the list. The *m\_updateCode* field will be used in Chapter 6 to determine if the collection has changed while the user is iterating over it. It is easier to add it to the code now instead of changing the code in Chapter 6.

## Creating Constructors

The `SingleLinkedList(T)` class will contain two constructors. One constructor is for creating an empty class, and the other is for creating a class with default values. These constructors are defined as follows.

C#

```
/// <summary>
/// Initializes a new instance of the SingleLinkedList(T) class that is empty.
/// </summary>
public SingleLinkedList()
{
}

/// <summary>
/// Initializes a new instance of the SingleLinkedList(T) class that
/// contains the items in the array.
/// </summary>
/// <param name="items">Adds the items to the end of the SingleLinkedList(T).</param>
public SingleLinkedList(IEnumerable<T> items)
{
    foreach (T item in items)
    {
        AddToEnd(item);
    }
}
```

Visual Basic

```
''' <summary>
''' Initializes a new instance of the SingleLinkedList(T) class that is empty.
''' </summary>
Public Sub New()
End Sub

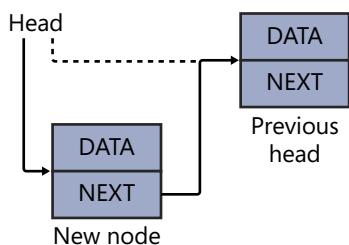
''' <summary>
''' Initializes a new instance of the SingleLinkedList(T) class that
''' contains the items in the list.
''' </summary>
''' <param name="items">Adds the items to the end of the SingleLinkedList(T).</param>
Public Sub New(ByVal items As IEnumerable(Of T))
    For Each item As T In items
        AddToEnd(item)
    Next
End Sub
```

The second constructor adds the specified items to the end of the list in the order they are in `items`.

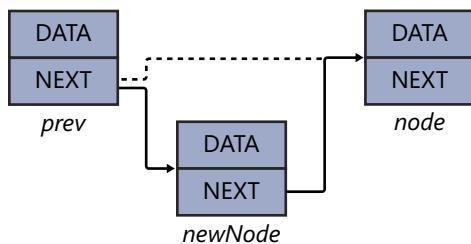
## Allowing Users to Add Items

Users can add items to a collection before or after a node or at the beginning or end of the list. This can be explained as three different conditions: adding to the beginning, adding to the middle, and adding to the end.

First, let's look at adding to the beginning. This condition happens if the user adds to an empty list, calls the *AddBefore* with the head node specified, or calls the *AddToBeginning* method. To accomplish this type of add, you need to set the newly added node's *Next* property to the previous head. You then need to assign the *Head* property to the newly added node. The *Count* must then be incremented to reflect the newly added node. The dashed line in the following illustration shows the link to the previous head.

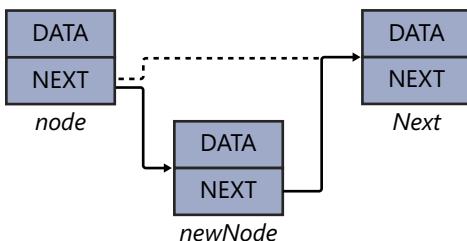


Next, let's look at adding to the middle. This condition happens if the user calls *AddAfter* or *AddBefore* without the head or tail node. When this happens, the node before the newly added node must be updated as well. You will have a reference to that node in the *AddAfter* method but not in the *AddBefore* method. For the *AddBefore* method, you need to traverse the list, while maintaining the node before the current node as the variable *prev*, until you reach the node you are adding before. You then need to assign the newly added node's *Next* property to the *Next* property of *prev*. Finally, you need to set the *Next* property of *prev* to the newly added node. The dashed line represents the references before the operation. The node you are adding before is denoted as *node*, and the new node is denoted as *newNode* in the following illustration and in the source code.



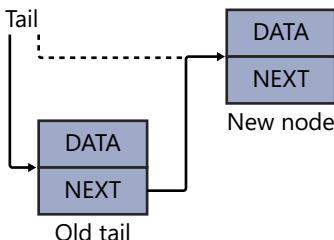
For the *AddAfter* method, you need to assign the newly added node's *Next* property to the *Next* property value of the node you are adding after. You then need to assign the *Next* property of the node you are adding after to the newly added node. The *Count* must then be incremented to reflect the newly added node. The dashed line represents the references

before the operation. The node you are adding after is denoted as *node*, and the new node is denoted as *newNode* in the following illustration and in the source code. The original node set as the *Next* property of *node* is denoted as *Next* in the following illustration.



As you can see, *AddAfter* is a lot more efficient than *AddBefore* because you do not have to traverse the list for an *AddAfter*. For both methods, you need to increment the *Count*.

Finally, look at adding to the end. This condition happens if the user calls the *AddToEnd* method or the *AddAfter* method with the tail node. To accomplish this type of add, set the previous tail's *Next* property to the newly added node. You then need to assign the *Tail* property to the newly added node. The *Count* must then be incremented to reflect the newly added node. The dashed line in the following illustration shows the link to the previous tail.



Now let's look at the implementation of these methods. Adding to the beginning and adding to the end can be executed using the following methods.

C#

```

/// <summary>
/// Adds the value to the beginning of the SingleLinkedList(T).
/// </summary>
/// <param name="value">
/// The value to add to the beginning of the SingleLinkedList(T).
/// </param>
/// <returns>The newly created node that is holding the value.</returns>
public SingleLinkedListNode<T> AddToBeginning(T value)
{
    SingleLinkedListNode<T> newNode = new SingleLinkedListNode<T>(this, value);

    if (IsEmpty)

```

```

    {
        m_head = newNode;
        m_tail = newNode;
    }
    else
    {
        newNode.Next = m_head;
        m_head = newNode;
    }

    ++m_count;
    ++m_updateCode;

    return newNode;
}

/// <summary>
/// Adds the value to the end of the SingleLinkedList(T).
/// </summary>
/// <param name="value">The value to add to the end of the SingleLinkedList(T).</param>
/// <returns>The newly created node that is holding the value.</returns>
public SingleLinkedListNode<T> AddToEnd(T value)
{
    SingleLinkedListNode<T> newNode = new SingleLinkedListNode<T>(this, value);

    if (IsEmpty)
    {
        m_head = newNode;
        m_tail = newNode;
    }
    else
    {
        m_tail.Next = newNode;
        m_tail = newNode;
    }

    ++m_count;
    ++m_updateCode;

    return newNode;
}

```

### Visual Basic

```

''' <summary>
''' Adds the value to the beginning of the SingleLinkedListNode(T).
''' </summary>
''' <param name="value">
''' The value to add to the beginning of the SingleLinkedListNode(T).
''' </param>
''' <returns>The newly created node that is holding the value.</returns>
Public Function AddToBeginning(ByVal value As T) As SingleLinkedListNode(Of T)
    Dim newNode As New SingleLinkedListNode(Of T)(Me, value)

```

```
If (IsEmpty) Then
    m_head = newNode
    m_tail = newNode
Else
    newNode.Next = m_head
    m_head = newNode
End If

m_count += 1
m_updateCode += 1

Return newNode
End Function

''' <summary>
''' Adds the value to the end of the SingleLinkedListNode(T).
''' </summary>
''' <param name="value">The value to add to the end of the SingleLinkedListNode(T).</param>
''' <returns>The newly created node that is holding the value.</returns>
Public Function AddToEnd(ByVal value As T) As SingleLinkedListNode(Of T)
    Dim newNode As New SingleLinkedListNode(Of T)(Me, value)

    If (IsEmpty) Then
        m_head = newNode
        m_tail = newNode
    Else
        m_tail.Next = newNode
        m_tail = newNode
    End If

    m_count += 1
    m_updateCode += 1

    Return newNode
End Function
```



**Note** Using the field *m\_tail* eliminates the need to traverse the complete list to find the last node. Adding an item increases the memory size of the list but eliminates the constant need to traverse the collection to add the item. Traversing the complete list to do an add may not be a big deal with a one-time add to a list with 10 items, but it is very beneficial in applications that add items frequently.

When you are adding to the middle of the list, you use the *AddAfter* and *AddBefore* methods. Each method will have an overload that allows the user to specify a value or a node to add. If the user specifies a value, a node will be returned that holds the value the user passed in. The implementation of these methods is as follows.

### C#

```
''' <summary>
''' Adds the specified value to the SingleLinkedList(T) after the specified node.
''' </summary>
```

```
/// <param name="node">The node to add the value after.</param>
/// <param name="value">The value to add.</param>
/// <returns>The newly created node that holds the value.</returns>
public SingleLinkedListNode<T> AddAfter(SingleLinkedListNode<T> node, T value)
{
    SingleLinkedListNode<T> newNode = new SingleLinkedListNode<T>(this, value);
    AddAfter(node, newNode);
    return newNode;
}

/// <summary>
/// Adds the specified newNode to the SingleLinkedList(T) after the specified node.
/// </summary>
/// <param name="node">The node to add the newNode after.</param>
/// <param name="newNode">The node to add.</param>
public void AddAfter(SingleLinkedListNode<T> node, SingleLinkedListNode<T> newNode)
{
    if (node == null)
    {
        throw new ArgumentNullException("node");
    }
    if (newNode == null)
    {
        throw new ArgumentNullException("newNode");
    }
    if (node.Owner != this)
    {
        throw new InvalidOperationException("node is not owned by this list");
    }
    if (newNode.Owner != this)
    {
        throw new InvalidOperationException("newNode is not owned by this list");
    }

    // The newly added node becomes the tail if you are adding after the tail
    if (m_tail == node)
    {
        m_tail = newNode;
    }

    newNode.Next = node.Next;
    node.Next = newNode;
    ++m_count;
    ++m_updateCode;
}

/// <summary>
/// Adds the specified value to the SingleLinkedList(T) before the specified node.
/// </summary>
/// <param name="node">The node to add the value before.</param>
/// <param name="value">The value to add.</param>
/// <returns>The newly created node that holds the value.</returns>
```

```
public SingleLinkedListNode<T> AddBefore(SingleLinkedListNode<T> node, T value)
{
    SingleLinkedListNode<T> newNode = new SingleLinkedListNode<T>(this, value);
    AddBefore(node, newNode);
    return newNode;
}

/// <summary>
/// Adds the specified newNode to the SingleLinkedList(T) before the specified node.
/// </summary>
/// <param name="node">The node to add the newNode before.</param>
/// <param name="newNode">The node to add.</param>
public void AddBefore(SingleLinkedListNode<T> node, SingleLinkedListNode<T> newNode)
{
    if (node == null)
    {
        throw new ArgumentNullException("node");
    }
    if (newNode == null)
    {
        throw new ArgumentNullException("newNode");
    }
    if (node.Owner != this)
    {
        throw new InvalidOperationException("node is not owned by this list");
    }
    if (newNode.Owner != this)
    {
        throw new InvalidOperationException("newNode is not owned by this list");
    }

    if (m_head == node)
    {
        newNode.Next = m_head;
        m_head = newNode;
    }
    else
    {
        // We have to find the node before the one we are inserting in front of

        SingleLinkedListNode<T> beforeNode = m_head;

        while (beforeNode != null && beforeNode.Next != node)
        {
            beforeNode = beforeNode.Next;
        }

        // We should always find node in the list
        if (beforeNode == null)
        {
            throw new InvalidOperationException("Something went wrong");
        }
    }
}
```

```

        newNode.Next = node;
        beforeNode.Next = newNode;
    }

    ++m_count;
    ++m_updateCode;
}

```

### Visual Basic

```

''' <summary>
''' Adds the specified value to the SingleLinkedList(T) after the specified node.
''' </summary>
''' <param name="node">The node to add the value after.</param>
''' <param name="value">The value to add.</param>
''' <returns>The newly created node that holds the value.</returns>
Public Function AddAfter(ByVal node As SingleLinkedListNode(Of T), ByVal value As T) _
    As SingleLinkedListNode(Of T)
    Dim newNode As New SingleLinkedListNode(Of T)(Me, value)
    AddAfter(node, newNode)
    Return newNode
End Function

''' <summary>
''' Adds the specified newNode to the SingleLinkedList(T) after the specified node.
''' </summary>
''' <param name="node">The node to add the newNode after.</param>
''' <param name="newNode">The node to add.</param>
Public Sub AddAfter(ByVal node As SingleLinkedListNode(Of T), ByVal newNode _ 
    As SingleLinkedListNode(Of T))
    If (node Is Nothing) Then
        Throw New ArgumentNullException("node")
    End If
    If (newNode Is Nothing) Then
        Throw New ArgumentNullException("newNode")
    End If
    If (node.Owner IsNot Me) Then
        Throw New InvalidOperationException("node is not owned by this list")
    End If
    If (newNode.Owner IsNot Me) Then
        Throw New InvalidOperationException("newNode is not owned by this list")
    End If

    ' The newly added node becomes the tail if you are adding after the tail
    If (node Is m_tail) Then
        m_tail = newNode
    End If

    newNode.Next = node.Next
    node.Next = newNode
    m_count += 1
    m_updateCode += 1
End Sub

```

```
''' <summary>
''' Adds the specified value to the SingleLinkedList(T) before the specified node.
''' </summary>
''' <param name="node">The node to add the value before.</param>
''' <param name="value">The value to add.</param>
''' <returns>The newly created node that holds the value.</returns>
Public Function AddBefore(ByVal node As SingleLinkedListNode(Of T), ByVal value As T) _
    As SingleLinkedListNode(Of T)
    Dim newNode As New SingleLinkedListNode(Of T)(Me, value)
    AddBefore(node, newNode)
    Return newNode
End Function

''' <summary>
''' Adds the specified newNode to the SingleLinkedList(T) before the specified node.
''' </summary>
''' <param name="node">The node to add the newNode before.</param>
''' <param name="newNode">The node to add.</param>
Public Sub AddBefore(ByVal node As SingleLinkedListNode(Of T), ByVal newNode _ 
    As SingleLinkedListNode(Of T))
    If (node Is Nothing) Then
        Throw New ArgumentNullException("node")
    End If
    If (newNode Is Nothing) Then
        Throw New ArgumentNullException("newNode")
    End If
    If (node.Owner IsNot Me) Then
        Throw New InvalidOperationException("node is not owned by this list")
    End If
    If (newNode.Owner IsNot Me) Then
        Throw New InvalidOperationException("newNode is not owned by this list")
    End If

    If (m_head Is node) Then
        newNode.Next = m_head
        m_head = newNode
    Else
        ' We have to find the node before the one we are inserting in front of

        Dim beforeNode As SingleLinkedListNode(Of T) = m_head

        While (Not beforeNode Is Nothing)
            If (beforeNode.Next Is node) Then
                Exit While
            End If
            beforeNode = beforeNode.Next
        End While
        ' We should always find node in the list
        If (beforeNode Is Nothing) Then
            Throw New InvalidOperationException("Something went wrong")
        End If
    End If
End Sub
```

```

        newNode.Next = node
        beforeNode.Next = newNode
    End If

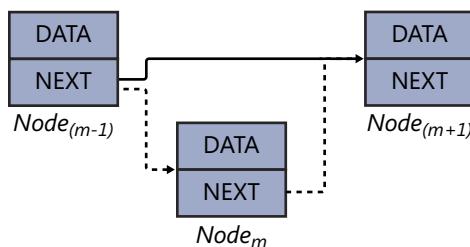
    m_count += 1
    m_updateCode += 1
End Sub

```

The code does some basic error checking.

## Allowing Users to Remove Items

When a *Remove* method is executed, the node before and after the node to remove needs to be linked together. If the removed node is the head, the removed node's *Next* property needs to be assigned to the *Head* property. If the removed node is the tail and the list isn't empty, the list will need to be traversed to find the node located before the removed node, as described previously in the *AddBefore* method. The node before the removed node will then be assigned to the *Tail* property. If the removed *Node<sub>m</sub>* is located between *Node<sub>(m-1)</sub>* and *Node<sub>(m+1)</sub>*, the nodes before and after the removed node are linked as shown in the following illustration.



### C#

```

/// <summary>
/// Removes the first occurrence of the specified item from the SingleLinkedList(T).
/// </summary>
/// <param name="item">The item to remove from the SingleLinkedList(T).</param>
/// <returns>True if an item was removed, false otherwise.</returns>
public bool Remove(T item)
{
    return Remove(item, false);
}

/// <summary>
/// Removes the first or all occurrences of the specified item from the SingleLinkedList(T).
/// </summary>
/// <param name="item">The item to remove from the SingleLinkedList(T).</param>
/// <param name="allOccurrences">
/// True if all nodes should be removed that contain the specified item,
/// False otherwise
/// </param>
/// <returns>True if an item was removed, false otherwise.</returns>

```

```
public bool Remove(T item, bool allOccurrences)
{
    if (IsEmpty)
    {
        return false;
    }

    SingleLinkedListNode<T> prev = null;
    SingleLinkedListNode<T> curr = Head;
    bool removed = false;

    EqualityComparer<T> comparer = EqualityComparer<T>.Default;

    // Start traversing the list at the head
    while (curr != null)
    {
        // Check to see if the current node contains the data we are trying to delete
        if (!comparer.Equals(curr.Data, item))
        {
            // Assign the current node to the previous node and
            // the previous node to the current node
            prev = curr;
            curr = curr.Next;
            continue;
        }

        // Create a pointer to the next node in the previous node
        if (prev != null)
        {
            prev.Next = curr.Next;
        }

        if (curr == Head)
        {
            // If the current node is the head we will have to assign
            // the next node as the head
            Head = curr.Next;
        }

        if (curr == Tail)
        {
            // If the current node is the tail we will have to assign
            // the previous node as the tail
            Tail = prev;
        }

        // Save the pointer for clean up later
        SingleLinkedListNode<T> tmp = curr;

        // Advance the current to the next node
        curr = curr.Next;

        // Since the node will no longer be used clean up the pointers in it
        tmp.Next = null;
        tmp.Owner = null;
    }
}
```

```
// Decrement the counter since we have removed a node
--m_count;

removed = true;

if (!allOccurrences)
{
    break;
}

if (removed)
{
    ++m_updateCode;
}

return removed;
}

/// <summary>
/// Removes the specified node from the SingleLinkedList(T).
/// </summary>
/// <param name="node">The node to remove from the SingleLinkedList(T).</param>
public void Remove(SingleLinkedListNode<T> node)
{
    if (IsEmpty)
    {
        return;
    }

    if (node == null)
    {
        throw new ArgumentNullException("node");
    }

    if (node.Owner != this)
    {
        throw new InvalidOperationException("The node doesn't belong to this list.");
    }

SingleLinkedListNode<T> prev = null;
SingleLinkedListNode<T> curr = Head;

// Find the node located before the specified node by traversing the list.
while (curr != null && curr != node)
{
    prev = curr;
    curr = curr.Next;
}

// The node has been found if the current node equals the node we are looking for
if (curr == node)
{
```

```

// Assign the head to the next node if the specified node is the head
if (m_head == node)
{
    m_head = node.Next;
}

// Assign the tail to the previous node if the specified node is the tail
if (m_tail == node)
{
    m_tail = prev;
}

// Set the previous node next reference to the removed node's next reference.
if (prev != null)
{
    prev.Next = curr.Next;
}

// Null out the removed node's next pointer to be safe.
node.Next = null;
node.Owner = null;

--m_count;
++m_updateCode;
}
}

```

### Visual Basic

```

''' <summary>
''' Removes the first occurrence of the specified item from the SingleLinkedList(T).
''' </summary>
''' <param name="item">The item to remove from the SingleLinkedList(T).</param>
''' <returns>True if an item was removed, false otherwise.</returns>
Public Function Remove(ByVal item As T) As Boolean
    Return Remove(item, False)
End Function

''' <summary>
''' Removes the first or all occurrences of the specified item from the SingleLinkedList(T).
''' </summary>
''' <param name="item">The item to remove from the SingleLinkedList(T).</param>
''' <param name="allOccurrences">
''' True if all nodes should be removed that contain the specified item, False otherwise
''' </param>
''' <returns>True if an item was removed, false otherwise.</returns>
Public Function Remove(ByVal item As T, ByVal allOccurrences As Boolean) As Boolean
    If (IsEmpty) Then
        Return False
    End If

    Dim prev As SingleLinkedListNode(Of T) = Nothing
    Dim curr As SingleLinkedListNode(Of T) = Head
    Dim removed As Boolean = False

```

```

Dim comparer As EqualityComparer(Of T) = EqualityComparer(Of T).Default

' Start traversing the list at the head
While (Not curr Is Nothing)
    ' Check to see if the current node contains the data we are trying to delete
    If (Not comparer.Equals(curr.Data, item)) Then
        ' Assign the current node to the previous node and
        ' the previous node to the current node
        prev = curr
        curr = curr.Next
        Continue While
    End If

    ' Create a pointer to the next node in the previous node
    If (Not prev Is Nothing) Then
        prev.Next = curr.Next
    End If

    If (curr Is Head) Then
        ' If the current node is the head we will have to assign
        ' the next node as the head
        Head = curr.Next
    End If

    If (curr Is Tail) Then
        ' If the current node is the tail we will have to assign
        ' the previous node as the tail
        Tail = prev
    End If

    ' Save the pointer for clean up later
    Dim tmp As SingleLinkedListNode(Of T) = curr

    ' Advance the current to the next node
    curr = curr.Next

    ' Since the node will no longer be used clean up the pointers in it
    tmp.Next = Nothing
    tmp.Owner = Nothing

    ' Decrement the counter since we have removed a node
    m_count -= 1

    removed = True

    If (Not allOccurrences) Then
        Exit While
    End If
End While

If (removed) Then
    m_updateCode += 1
End If

Return removed
End Function

```

```
''' <summary>
''' Removes the specified node from the SingleLinkedList(T).
''' </summary>
''' <param name="node">The node to remove from the SingleLinkedList(T).</param>
Public Sub Remove(ByVal node As SingleLinkedListNode(Of T))
    If (IsEmpty) Then
        Return
    End If

    If (node Is Nothing) Then
        Throw New ArgumentNullException("node")
    End If

    If (node.Owner IsNot Me) Then
        Throw New InvalidOperationException("The node doesn't belong to this list.")
    End If

    Dim prev As SingleLinkedListNode(Of T) = Nothing
    Dim curr As SingleLinkedListNode(Of T) = Head

    ' Find the node located before the specified node by traversing the list.
    While (Not curr Is Nothing And curr IsNot node)
        prev = curr
        curr = curr.Next
    End While

    ' The node has been found if the current node equals the node we are looking for
    If (curr Is node) Then
        ' Assign the head to the next node if the specified node is the head
        If (m_head Is node) Then
            m_head = node.Next
        End If

        ' Assign the tail to the previous node if the specified node is the tail
        If (m_tail Is node) Then
            m_tail = prev
        End If

        ' Set the previous node next reference to the removed node's next reference.
        If (Not prev Is Nothing) Then
            prev.Next = curr.Next
        End If

        ' Null out the removed node's next pointer to be safe.
        node.Next = Nothing
        node.Owner = Nothing

        m_count -= 1
        m_updateCode += 1
    End If
End Sub
```

With the *Remove* methods, you can remove an item by using its node or value. If the value is specified, the method traverses the list until it finds the node that contains the value. The implementation for removing it is done as just described.

With the *Clear* method, the user can remove all nodes in the list without having to repeatedly call the *Remove* method with the *Head* pointer until the *Count* is 0.

### C#

```
/// <summary>
/// Removes all items from the SingleLinkedList(T).
/// </summary>
public void Clear()
{
    SingleLinkedListNode<T> tmp;

    // Clean up the items in the list
    for (SingleLinkedListNode<T> node = m_head; node != null; )
    {
        tmp = node.Next;

        // Change the count and head pointer in case we throw an exception.
        // this way the node is removed before we clear the data
        m_head = tmp;
        --m_count;

        // Erase the contents of the node
        node.Next = null;
        node.Owner = null;

        // Move to the next node
        node = tmp;
    }

    if (m_count <= 0)
    {
        m_head = null;
        m_tail = null;
    }

    ++m_updateCode;
}
```

### Visual Basic

```
''' <summary>
''' Removes all items from the SingleLinkedList(T).
''' </summary>
Public Sub Clear()
    Dim tmp As SingleLinkedListNode(Of T)

    ' Clean up the items in the list
    Dim node As SingleLinkedListNode(Of T) = m_head
    While (Not node Is Nothing)

        tmp = node.Next
```

```
' Change the count and head pointer in case we throw an exception.  
' this way the node is removed before we clear the data  
m_head = tmp  
m_count -= 1  
  
' Erase the contents of the node  
node.Next = Nothing  
node.Owner = Nothing  
  
' Move to the next node  
node = tmp  
End While  
  
If (m_count <= 0) Then  
    m_head = Nothing  
    m_tail = Nothing  
End If  
  
m_updateCode += 1  
End Sub
```

It helps to *null* out all of the nodes during a clear so that if any of the nodes are passed to you later, you can determine that the node no longer belongs to the collection. The preceding code updates the count and moves the head to the next node while it clears the current node. This is done because, if an error occurs during the clear, hopefully the collection will still point to somewhat valid data when accessed again. For example, if *Count* was equal to 12 and *m\_head* and *m\_tail* pointed to *null*, unpredictable things would happen while accessing some of the functions. Hopefully, the preceding example will point you in the right direction because it is not 100 percent bulletproof.

## Adding Helper Methods and Properties

Users will want the ability to check the status of the list. The *Contains* and *Find* methods let them look at the contents of the list, whereas the *Head*, *Tail*, *Count*, and *IsEmpty* properties allow them to look at the status of the list.

Your users may find it necessary to find the node of an item in the list or check to see if a node is present in the list. This information can stop them from having to unnecessarily traverse or operate on the list. The following method and properties allow users to find or check for the presence of a node.

The *Find* method allows users to find or check for the presence of a node. It locates a node in the list by searching for the value it contains, as follows.

### C#

```
/// <summary>  
/// Locates the first node that contains the specified data.  
/// </summary>  
/// <param name="data">The data to find.</param>  
/// <returns>The node that contains the specified data, null otherwise.</returns>
```

```

public SingleLinkedListNode<T> Find(T data)
{
    if (IsEmpty)
    {
        return null;
    }

    EqualityComparer<T> comparer = EqualityComparer<T>.Default;

    // Traverse the list from the Head to Tail.
    for (SingleLinkedListNode<T> curr = Head; curr != null; curr = curr.Next)
    {
        // Return the node we are currently on if it contains the data we are looking for.
        if (comparer.Equals(curr.Data, data))
        {
            return curr;
        }
    }

    return null;
}

```

### Visual Basic

```

''' <summary>
''' Locates the first node that contains the specified data.
''' </summary>
''' <param name="data">The data to find.</param>
''' <returns>The node that contains the specified data, null otherwise.</returns>
Public Function Find(ByVal data As T) As SingleLinkedListNode(Of T)
    If (IsEmpty) Then
        Return Nothing
    End If

    Dim comparer As EqualityComparer(Of T) = EqualityComparer(Of T).Default

    ' Traverse the list from the Head to Tail.
    Dim curr As SingleLinkedListNode(Of T) = Head
    While (Not curr Is Nothing)
        ' Return the node we are currently on if it contains the data we are looking for.
        If (comparer.Equals(curr.Data, data)) Then
            Return curr
        End If
        curr = curr.Next
    End While
    Return Nothing
End Function

```

The method traverses the list from the beginning until it finds the node that contains the data you are looking for. A *null* is returned if no node was found.

The *Contains* method is used to check to see if an item is present in the list.

**C#**

```
/// <summary>
/// Checks if the specified data is present in the SingleLinkedList(T).
/// </summary>
/// <param name="data">The data to look for.</param>
/// <returns>True if the data is found, false otherwise.</returns>
public bool Contains(T data)
{
    return Find(data) != null;
}
```

**Visual Basic**

```
''' <summary>
''' Checks if the specified data is present in the SingleLinkedList(T).
''' </summary>
''' <param name="data">The data to look for.</param>
''' <returns>True if the data is found, false otherwise.</returns>
Public Function Contains(ByVal data As T) As Boolean
    Return Not Find(data) Is Nothing
End Function
```

The method works by checking to see if the *Find* method found the specified node.

Users may also need to traverse the list for other reasons as well. To do this, they will need access to the head and tail.

**C#**

```
/// <summary>
/// Gets the head node in the SingleLinkedList(T).
/// </summary>
public SingleLinkedListNode<T> Head
{
    get { return m_head; }
    private set { m_head = value; }
}

/// <summary>
/// Gets the tail node in the SingleLinkedList(T).
/// </summary>
public SingleLinkedListNode<T> Tail
{
    get { return m_tail; }
    private set { m_tail = value; }
}
```

**Visual Basic**

```
''' <summary>
''' Gets the head node in the SingleLinkedList(T).
''' </summary>
Public Property Head() As SingleLinkedListNode(Of T)
    Get
        Return m_head
    End Get
```

```

    Private Set(ByVal value As SingleLinkedListNode(Of T))
        m_head = value
    End Set
End Property

''' <summary>
''' Gets the tail node in the SingleLinkedList(T).
''' </summary>
Public Property Tail() As SingleLinkedListNode(Of T)
    Get
        Return m_tail
    End Get

    Private Set(ByVal value As SingleLinkedListNode(Of T))
        m_tail = value
    End Set
End Property

```

The *Head* and *Tail* methods return the head and tail of the list respectively. Both properties block the setting of the values by the user.

Knowing how many items are in the list and if it is empty eliminates the need to perform operations on the list. The following properties help users determine what needs to be done.

### C#

```

/// <summary>
/// States if the SingleLinkedList(T) is empty.
/// </summary>
public bool IsEmpty
{
    get { return m_count <= 0; }
}

/// <summary>
/// Gets the number of nodes actually contained in the SingleLinkedList(T).
/// </summary>
public int Count
{
    get { return m_count; }
}

```

### Visual Basic

```

''' <summary>
''' States if the SingleLinkedList(T) is empty.
''' </summary>
Public ReadOnly Property IsEmpty() As Boolean
    Get
        Return m_count <= 0
    End Get
End Property

''' <summary>
''' Gets the number of nodes actually contained in the SingleLinkedList(T).
''' </summary>

```

```
Public ReadOnly Property Count() As Integer
    Get
        Return m_count
    End Get
End Property
```

The following helper function helps copy the contents of the linked list into a new array. The new array is useful when code requires an array instead of our class.

### C#

```
/// <summary>
/// Copies the elements of the SingleLinkedList<T> to a new array.
/// </summary>
/// <returns>
/// An array containing copies of the elements of the SingleLinkedList<T>.
/// </returns>
public T[] ToArray()
{
    T[] retval = new T[m_count];

    int index = 0;
    for (SingleLinkedListNode<T> i = Head; i != null; i = i.Next)
    {
        retval[index] = i.Data;
        ++index;
    }

    return retval;
}
```

### Visual Basic

```
''' <summary>
''' Copies the elements of the SingleLinkedList(T) to a new array.
''' </summary>
''' <returns>
''' An array containing copies of the elements of the SingleLinkedList(T).
''' </returns>
Public Function ToArray() As T()
    Dim retval As T() = New T(m_count - 1) {}

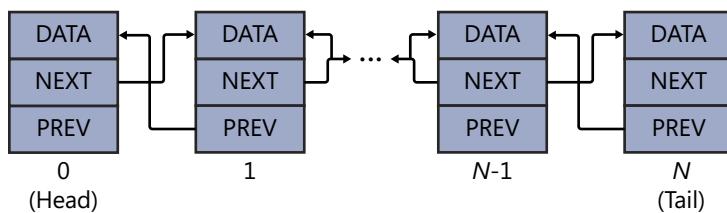
    Dim index As Integer = 0

    Dim i As SingleLinkedListNode(Of T) = Head
    While (Not i Is Nothing)
        retval(index) = i.Data
        index += 1
        i = i.Next
    End While
    Return retval
End Function
```

## Doubly Linked List Implementation

In a doubly linked list, each node contains a reference to the next node as well as to the previous node. The addition of the previous pointer eliminates some of the performance problems of a singly linked list by eliminating the need to search the list for the previous node. The additional pointer does increase the memory footprint of the list but enables you to traverse the list forward and backward from any node.

A doubly linked list is a sequence of nodes that reference the next and previous node. Each node contains data and a reference to the previous and next node.



### Creating the Node Class

As described earlier, the first node is called the head and the last node is called the tail. Each node is represented by a class called *DoubleLinkedListNode(T)*. The *DoubleLinkedListNode(T)* class is defined as follows.

**C#**

```

/// <summary>
/// Represents a node in a DoubleLinkedList(T).
/// </summary>
/// <typeparam name="T">Specifies the type of data in the node.</typeparam>
[DebuggerDisplay("Data={Data}")]
public class DoubleLinkedListNode<T>
{
    DoubleLinkedList<T> m_owner;
    DoubleLinkedListNode<T> m_prev;
    DoubleLinkedListNode<T> m_next;
    T m_data;

    /// <summary>
    /// Initializes a new instance of the DoubleLinkedListNode(T) class
    /// with the specified data.
    /// </summary>
    /// <param name="data">The data that this node will contain.</param>
    public DoubleLinkedListNode(T data)
    {
        m_data = data;
        m_owner = null;
    }
}

```

```
/// <summary>
/// Initializes a new instance of the DoubleLinkedListNode(T) class
/// with the specified data and owner.
/// </summary>
/// <param name="data">The data that this node will contain.</param>
internal DoubleLinkedListNode(DoubleLinkedList<T> owner, T data)
{
    m_data = data;
    m_owner = owner;
}

/// <summary>
/// Gets the next node.
/// </summary>
public DoubleLinkedListNode<T> Next
{
    get { return m_next; }
    internal set { m_next = value; }
}

/// <summary>
/// Gets or sets the owner of the node.
/// </summary>
internal DoubleLinkedList<T> Owner
{
    get { return m_owner; }
    set { m_owner = value; }
}

/// <summary>
/// Gets the previous node.
/// </summary>
public DoubleLinkedListNode<T> Previous
{
    get { return m_prev; }
    internal set { m_prev = value; }
}

/// <summary>
/// Gets the data contained in the node.
/// </summary>
public T Data
{
    get { return m_data; }
    internal set { m_data = value; }
}
```

### Visual Basic

```
''' <summary>
''' Represents a node in a DoubleLinkedList(T).
```

```
''' </summary>
''' <typeparam name="T">Specifies the type of data in the node.</typeparam>
<DebuggerDisplay("Data={Data}")> _
Public Class DoubleLinkedListNode(Of T)
    Private m_owner As DoubleLinkedList(Of T)
    Private m_prev As DoubleLinkedListNode(Of T)
    Private m_next As DoubleLinkedListNode(Of T)
    Private m_data As T

    ''' <summary>
    ''' Initializes a new instance of the DoubleLinkedListNode(T) class
    ''' with the specified data.
    ''' </summary>
    ''' <param name="data">The data that this node will contain.</param>
    Public Sub New(ByVal data As T)
        m_data = data
        m_owner = Nothing
    End Sub

    ''' <summary>
    ''' Initializes a new instance of the DoubleLinkedListNode(T) class
    ''' with the specified data and owner.
    ''' </summary>
    ''' <param name="data">The data that this node will contain.</param>
    Friend Sub New(ByVal owner As DoubleLinkedList(Of T), ByVal data As T)
        m_data = data
        m_owner = owner
    End Sub

    ''' <summary>
    ''' Gets the next node.
    ''' </summary>
    Public Property [Next]() As DoubleLinkedListNode(Of T)
        Get
            Return m_next
        End Get
        Friend Set(ByVal value As DoubleLinkedListNode(Of T))
            m_next = value
        End Set
    End Property

    ''' <summary>
    ''' Gets or sets the owner of the node.
    ''' </summary>
    Friend Property Owner() As DoubleLinkedList(Of T)
        Get
            Return m_owner
        End Get
        Set(ByVal value As DoubleLinkedList(Of T))
            m_owner = value
        End Set
    End Property
```

```
''' <summary>
''' Gets the previous node.
''' </summary>
Public Property Previous() As DoubleLinkedListNode(Of T)
    Get
        Return m_prev
    End Get
    Friend Set(ByVal value As DoubleLinkedListNode(Of T))
        m_prev = value
    End Set
End Property

''' <summary>
''' Gets the data contained in the node.
''' </summary>
Public Property Data() As T
    Get
        Return m_data
    End Get
    Friend Set(ByVal value As T)
        m_data = value
    End Set
End Property

End Class
```

The class is defined as a generic class, so the contained data will remain type-safe. The field *m\_data* contains the data that you want to store. The field *m\_next* references the next node in the list whereas the field *m\_prev* references the previous node in the list. The set property that wraps *m\_next* and *m\_prev* is internal so that users can't change the *next* and *prev* pointers and bypass the bookkeeping. The field *m\_owner* is used to verify that the node belongs to the list that is being operated on.

## Declaring the *DoubleLinkedList(T)* Class

The *DoubleLinkedList(T)* class is implemented as a doubly linked list and is defined as follows.

C#

```
[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(ArrayDebugView))]
public class DoubleLinkedList<T>
{
    // Fields
    private int m_count;
    private DoubleLinkedListNode<T> m_head;
    private DoubleLinkedListNode<T> m_tail;
    private int m_updateCode;

    // Constructors
    public DoubleLinkedList();
    public DoubleLinkedList(IEnumerable<T> items);
```

```

// Methods
public void AddAfter(DoubleLinkedListNode<T> node, DoubleLinkedListNode<T> newNode);
public DoubleLinkedListNode<T> AddAfter(DoubleLinkedListNode<T> node, T value);
public void AddBefore(DoubleLinkedListNode<T> node, DoubleLinkedListNode<T> newNode);
public DoubleLinkedListNode<T> AddBefore(DoubleLinkedListNode<T> node, T value);
public DoubleLinkedListNode<T> AddToBeginning(T value);
public DoubleLinkedListNode<T> AddToEnd(T value);
public void Clear();
public bool Contains(T data);
public DoubleLinkedListNode<T> Find(T data);
public bool Remove(T item);
public void Remove(DoubleLinkedListNode<T> node);
public bool Remove(T item, bool allOccurrences);
public T[] ToArray();
public T[] ToArrayReversed();

// Properties
public int Count { get; }
public DoubleLinkedListNode<T> Head { get; private set; }
public bool IsEmpty { get; }
public DoubleLinkedListNode<T> Tail { get; private set; }
}

```

### Visual Basic

```

<DebuggerTypeProxy(GetType(ArrayDebugView))> _
<DebuggerDisplay("Count={Count}")> _
Public Class DoubleLinkedList(Of T)
    ' Fields
    Private m_count As Integer
    Private m_head As DoubleLinkedListNode(Of T)
    Private m_tail As DoubleLinkedListNode(Of T)
    Private m_updateCode As Integer

    ' Constructors
    Public Sub New()
        Public Sub New(ByVal items As IEnumerable(Of T))

    ' Methods
    Public Function AddAfter(ByVal node As DoubleLinkedListNode(Of T), ByVal value As T) As DoubleLinkedListNode(Of T)
        Public Sub AddAfter(ByVal node As DoubleLinkedListNode(Of T), ByVal newNode As DoubleLinkedListNode(Of T))
        Public Sub AddBefore(ByVal node As DoubleLinkedListNode(Of T), ByVal newNode As DoubleLinkedListNode(Of T))
        Public Function AddBefore(ByVal node As DoubleLinkedListNode(Of T), ByVal value As T) As DoubleLinkedListNode(Of T)
        Public Function AddToBeginning(ByVal value As T) As DoubleLinkedListNode(Of T)
        Public Function AddToEnd(ByVal value As T) As DoubleLinkedListNode(Of T)
        Public Sub Clear()
        Public Function Contains(ByVal data As T) As Boolean
        Public Function Find(ByVal data As T) As DoubleLinkedListNode(Of T)
        Public Function Remove(ByVal item As T) As Boolean
        Public Sub Remove(ByVal node As DoubleLinkedListNode(Of T))

```

```
Public Function Remove(ByVal item As T, ByVal allOccurrences As Boolean) As Boolean
Public Function ToArray() As T()
Public Function ToArrayReversed() As T()

' Properties
Public ReadOnly Property Count As Integer
Property Head As DoubleLinkedListNode(Of T)
Public ReadOnly Property IsEmpty As Boolean
Property Tail As DoubleLinkedListNode(Of T)

End Class
```

For performance reasons, the class maintains a reference to the last node in the field *m\_tail*. The *m\_tail* field provides quick access to the last item when a user wants to add a node to the list. This eliminates the need to traverse the list to find the last node. The field *m\_head* references the first node, and the field *m\_count* tracks the number of nodes in the list.

The *m\_updateCode* field will be incremented each time the user modifies the list. The *m\_updateCode* field will be used in Chapter 6 to determine if the collection has changed while the user is iterating over it. It is easier to add it to the code now instead of changing the code in Chapter 6.

## Creating Constructors

The *DoubleLinkedList(T)* class will contain two constructors. One constructor is for creating an empty class, and the other is for creating a class with default values. These constructors are defined as follows.

### C#

```
/// <summary>
/// Initializes a new instance of the DoubleLinkedList (T) class that is empty.
/// </summary>
public DoubleLinkedList()
{
}

/// <summary>
/// Initializes a new instance of the DoubleLinkedList (T) class
/// that contains the items in the list.
/// </summary>
/// <param name="items">Adds the items to the end of the DoubleLinkedList (T).</param>
public DoubleLinkedList (IEnumerable<T> items)
{
    foreach (T item in items)
    {
        AddToEnd(item);
    }
}
```

### Visual Basic

```

''' <summary>
''' Initializes a new instance of the DoubleLinkedList(T) class that is empty.
''' </summary>
Public Sub New()
End Sub

''' <summary>
''' Initializes a new instance of the DoubleLinkedList(T) class
''' that contains the items in the list.
''' </summary>
''' <param name="items">Adds the items to the end of the DoubleLinkedList(T).</param>
Public Sub New(ByVal items As IEnumerable(Of T))
    For Each item As T In items
        AddToEnd(item)
    Next
End Sub

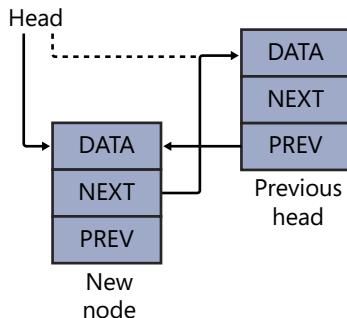
```

The second constructor adds the specified items to the end of the list in the order they are in *items*.

## Allowing Users to Add Items

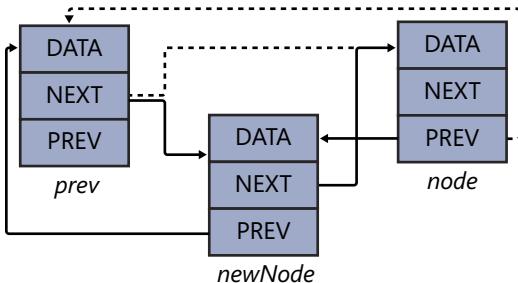
Users can add items to a collection before or after a node or at the beginning or end of the list. This can be explained as three different conditions: adding to the beginning, adding to the middle, and adding to the end.

First, let's look at adding to the beginning. This condition happens if the user adds to an empty list, calls the *AddBefore* method with the head node specified, or calls the *AddToBeginning* method. To accomplish this type of add, you need to set the newly added node's *Next* property to the previous head. Then, assign the previous head's *Previous* property to the newly added node. Also, you need to assign the *Head* property to the newly added node. The *Count* property must then be incremented to reflect the newly added node. The dashed line in the following illustration shows the link to the previous head.

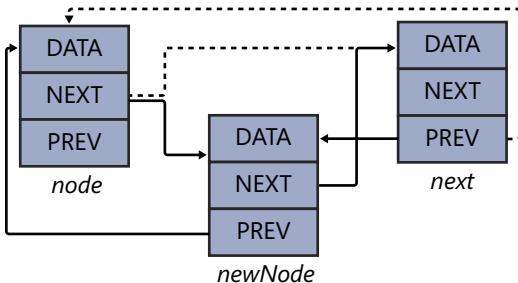


Next, let's look at adding to the middle. This condition happens if the user calls *AddAfter* or *AddBefore* without the head or tail node. First, assign the newly added node's *Next* property

to the node you are adding before. The newly added node's *Prev* property should be set to the node you are adding after. The *Prev* property of the node you are adding before and the *Next* property of the node you are adding after needs to be assigned to the newly added node. The dashed line represents the references before the operation. The node you are adding before is denoted as *node* and the new node is denoted as *newNode* in the following illustration and in the source code. The original node set as the *Prev* property of *node* is denoted as *prev* in the following illustration.



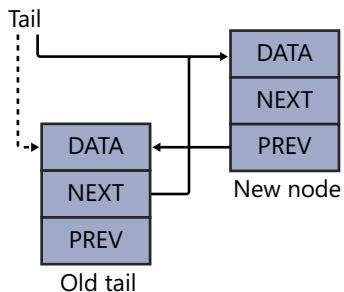
For the *AddAfter* method, you need to assign the newly added node's *Next* property to the *Next* property of the node you are adding after. Then you need to assign the new node's *Prev* property to the node you are adding after. The original node after the node you are adding needs to have its *Prev* property set to the newly added node. You then need to assign the *Next* property of the node you are adding after to the newly added node. The dashed line represents the references before the operation. The node you are adding after is denoted as *node* and the new node is denoted as *newNode* in the following illustration and in the source code. The original node set as the *Next* property of *node* is denoted as *next* in the following illustration.



The *AddBefore* in a doubly linked list is far more efficient than the one in a singly linked list, because you already have the previous node in a doubly linked list.

Last, let's look at adding to the end. This condition happens if the user calls the *AddToEnd* method or the *AddAfter* method with the tail node. To accomplish this type of add, you need to set the previous tail's *Next* property to the newly added node. Next, you need to assign the new node's *Previous* property to the newly added node. You then need to assign the *Tail*

property to the newly added node. The *Count* must then be incremented to reflect the newly added node. The dashed line in the following illustration shows the link to the previous tail.



Now let's look at the implementation of these methods. Adding to the beginning and adding to the end can be executed using the following methods.

**C#**

```

/// <summary>
/// Adds the value to the beginning of the DoubleLinkedList(T).
/// </summary>
/// <param name="value">
/// The value to add to the beginning of the DoubleLinkedList(T).
/// </param>
/// <returns>The newly created node that is holding the value.</returns>
public DoubleLinkedListNode<T> AddToBeginning(T value)
{
    DoubleLinkedListNode<T> newNode = new DoubleLinkedListNode<T>(this, value);

    if (IsEmpty)
    {
        m_head = newNode;
        m_tail = newNode;
    }
    else
    {
        newNode.Next = m_head;
        m_head.Previous = newNode;
        m_head = newNode;
    }

    ++m_count;
    ++m_updateCode;

    return newNode;
}

/// <summary>
/// Adds the value to the end of the DoubleLinkedList(T).
/// </summary>
/// <param name="value">The value to add to the end of the DoubleLinkedList(T).</param>
/// <returns>The newly created node that is holding the value.</returns>
  
```

```

public DoubleLinkedListNode<T> AddToEnd(T value)
{
    DoubleLinkedListNode<T> newNode = new DoubleLinkedListNode<T>(this, value);

    if (IsEmpty)
    {
        m_head = newNode;
        m_tail = newNode;
    }
    else
    {
        newNode.Previous = m_tail;
        m_tail.Next = newNode;
        m_tail = newNode;
    }

    ++m_count;
    ++m_updateCode;

    return newNode;
}

```

### Visual Basic

```

''' <summary>
''' Adds the value to the beginning of the DoubleLinkedList(T).
''' </summary>
''' <param name="value">
''' The value to add to the beginning of the DoubleLinkedList(T).
''' </param>
''' <returns>The newly created node that is holding the value.</returns>
Public Function AddToBeginning(ByVal value As T) As DoubleLinkedListNode(Of T)
    Dim newNode As New DoubleLinkedListNode(Of T)(Me, value)

    If (IsEmpty) Then
        m_head = newNode
        m_tail = newNode
    Else
        newNode.Next = m_head
        m_head.Previous = newNode
        m_head = newNode
    End If

    m_count += 1
    m_updateCode += 1

    Return newNode
End Function

''' <summary>
''' Adds the value to the end of the DoubleLinkedList(T).
''' </summary>
''' <param name="value">The value to add to the end of the DoubleLinkedList(T).</param>
''' <returns>The newly created node that is holding the value.</returns>

```

```
Public Function AddToEnd(ByVal value As T) As DoubleLinkedListNode(Of T)
    Dim newNode As New DoubleLinkedListNode(Of T)(Me, value)

    If (IsEmpty) Then
        m_head = newNode
        m_tail = newNode
    Else
        newNode.Previous = m_tail
        m_tail.Next = newNode
        m_tail = newNode
    End If

    m_count += 1
    m_updateCode += 1

    Return newNode
End Function
```



**Note** Using the field *m\_tail* eliminates the need to traverse the complete list to find the last node. Adding a field increases the memory size of the list but eliminates the constant need to traverse the collection to add the item. This may not be a big deal with a one-time add to a list with 10 items, but is very beneficial in applications that add items frequently.

When you are adding to the middle of the list, you will use the *AddAfter* and *AddBefore* methods. Each method will have an overload that allows the user to specify a value or a node to add. If the user specifies a value, a node will be returned that holds the value passed in. The implementation of these methods is as follows.

### C#

```
/// <summary>
/// Adds the specified value to the DoubleLinkedList(T) after the specified node.
/// </summary>
/// <param name="node">The node to add the value after.</param>
/// <param name="value">The value to add.</param>
/// <returns>The newly created node that holds the value.</returns>
public DoubleLinkedListNode<T> AddAfter(DoubleLinkedListNode<T> node, T value)
{
    DoubleLinkedListNode<T> newNode = new DoubleLinkedListNode<T>(this, value);
    AddAfter(node, newNode);
    return newNode;
}

/// <summary>
/// Adds the specified newNode to the DoubleLinkedList(T) after the specified node.
/// </summary>
/// <param name="node">The node to add the newNode after.</param>
/// <param name="newNode">The node to add.</param>
public void AddAfter(DoubleLinkedListNode<T> node, DoubleLinkedListNode<T> newNode)
```

```
{  
    if (node == null)  
    {  
        throw new ArgumentNullException("node");  
    }  
    if (newNode == null)  
    {  
        throw new ArgumentNullException("newNode");  
    }  
    if (node.Owner != this)  
    {  
        throw new InvalidOperationException("node is not owned by this list");  
    }  
    if (newNode.Owner != this)  
    {  
        throw new InvalidOperationException("newNode is not owned by this list");  
    }  
  
    if (node == m_tail)  
    {  
        m_tail = newNode;  
    }  
  
    if (node.Next != null)  
    {  
        node.Next.Previous = newNode;  
    }  
  
    newNode.Next = node.Next;  
    newNode.Previous = node;  
  
    node.Next = newNode;  
  
    ++m_count;  
    ++m_updateCode;  
}  
  
/// <summary>  
/// Adds the specified value to the DoubleLinkedList(T) before the specified node.  
/// </summary>  
/// <param name="node">The node to add the value before.</param>  
/// <param name="value">The value to add.</param>  
/// <returns>The newly created node that holds the value.</returns>  
public DoubleLinkedListNode<T> AddBefore(DoubleLinkedListNode<T> node, T value)  
{  
    DoubleLinkedListNode<T> newNode = new DoubleLinkedListNode<T>(this, value);  
    AddBefore(node, newNode);  
    return newNode;  
}  
  
/// <summary>  
/// Adds the specified newNode to the DoubleLinkedList(T) before the specified node.  
/// </summary>  
/// <param name="node">The node to add the newNode before.</param>  
/// <param name="newNode">The node to add.</param>
```

```

public void AddBefore(DoubleLinkedListNode<T> node, DoubleLinkedListNode<T> newNode)
{
    if (node == null)
    {
        throw new ArgumentNullException("node");
    }
    if (newNode == null)
    {
        throw new ArgumentNullException("newNode");
    }
    if (node.Owner != this)
    {
        throw new InvalidOperationException("node is not owned by this list");
    }
    if (newNode.Owner != this)
    {
        throw new InvalidOperationException("newNode is not owned by this list");
    }

    // We have to find the node before this one
    if (m_head == node)
    {
        newNode.Next = m_head;
        m_head.Previous = newNode;
        m_head = newNode;
    }
    else
    {
        // Set the node before the node we are inserting in front of Next to the new node
        if (node.Previous != null)
        {
            node.Previous.Next = newNode;
        }

        newNode.Previous = node.Previous;
        newNode.Next = node;

        node.Previous = newNode;
    }

    ++m_count;
    ++m_updateCode;
}

```

### Visual Basic

```

''' <summary>
''' Adds the specified value to the DoubleLinkedList(T) after the specified node.
''' </summary>
''' <param name="node">The node to add the value after.</param>
''' <param name="value">The value to add.</param>
''' <returns>The newly created node that holds the value.</returns>
Public Function AddAfter(ByVal node As DoubleLinkedListNode(Of T), ByVal value As T) _
    As DoubleLinkedListNode(Of T)
    Dim newNode As DoubleLinkedListNode(Of T) = New DoubleLinkedListNode(Of T)(Me, value)

```

```
    AddAfter(node, newNode)
    Return newNode
End Function

''' <summary>
''' Adds the specified newNode to the DoubleLinkedList(T) after the specified node.
''' </summary>
''' <param name="node">The node to add the newNode after.</param>
''' <param name="newNode">The node to add.</param>
Public Sub AddAfter( ByVal node As DoubleLinkedListNode(Of T) , ByVal newNode _ 
    As DoubleLinkedListNode(Of T))
    If (node Is Nothing) Then
        Throw New ArgumentNullException("node")
    End If
    If (newNode Is Nothing) Then
        Throw New ArgumentNullException("newNode")
    End If
    If (node.Owner IsNot Me) Then
        Throw New InvalidOperationException("node is not owned by this list")
    End If
    If (newNode.Owner IsNot Me) Then
        Throw New InvalidOperationException("newNode is not owned by this list")
    End If

    If (node Is m_tail) Then
        m_tail = newNode
    End If

    If (node.Next IsNot Nothing) Then
        node.Next.Previous = newNode
    End If

    newNode.Next = node.Next
    newNode.Previous = node

    node.Next = newNode

    m_count += 1
    m_updateCode += 1
End Sub

''' <summary>
''' Adds the specified value to the DoubleLinkedList(T) before the specified node.
''' </summary>
''' <param name="node">The node to add the value before.</param>
''' <param name="value">The value to add.</param>
''' <returns>The newly created node that holds the value.</returns>
Public Function AddBefore( ByVal node As DoubleLinkedListNode(Of T) , ByVal value As T ) _ 
    As DoubleLinkedListNode(Of T)
    Dim newNode As DoubleLinkedListNode(Of T) = New DoubleLinkedListNode(Of T)(Me, value)
    AddBefore(node, newNode)
    Return newNode
End Function
```

```

''' <summary>
''' Adds the specified newNode to the DoubleLinkedList(T) before the specified node.
''' </summary>
''' <param name="node">The node to add the newNode before.</param>
''' <param name="newNode">The node to add.</param>
Public Sub AddBefore(ByVal node As DoubleLinkedListNode(Of T), ByVal newNode _
    As DoubleLinkedListNode(Of T))
    If (node Is Nothing) Then
        Throw New ArgumentNullException("node")
    End If
    If (newNode Is Nothing) Then
        Throw New ArgumentNullException("newNode")
    End If
    If (node.Owner IsNot Me) Then
        Throw New InvalidOperationException("node is not owned by this list")
    End If
    If (newNode.Owner IsNot Me) Then
        Throw New InvalidOperationException("newNode is not owned by this list")
    End If

    ' We have to find the node before this one
    If (m_head Is node) Then
        newNode.Next = m_head
        m_head.Previous = newNode
        m_head = newNode
    Else
        ' Set the node before the node we are inserting in front of Next to the new node
        If (node.Previous IsNot Nothing) Then
            node.Previous.Next = newNode
        End If

        newNode.Previous = node.Previous
        newNode.Next = node

        node.Previous = newNode
    End If

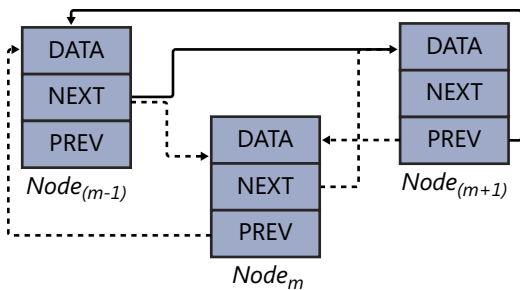
    m_count += 1
    m_updateCode += 1
End Sub

```

The code will do some basic error checking.

## Allowing Users to Remove Items

When a remove is executed, the nodes before and after the node that needs to be removed will need to be linked together. If the removed node is the head, the next node will need to be assigned to *Head*. If the removed node is the tail and the list isn't empty, the previous node will need to be assigned to *Tail*. If the removed *Node<sub>m</sub>* is located between two nodes, the nodes before and after the removed *Node<sub>(m-1)</sub>* and *Node<sub>(m+1)</sub>* are linked, as shown in the following illustration.



C#

```

/// <summary>
/// Removes the first occurrence of the specified item from the DoubleLinkedList(T).
/// </summary>
/// <param name="item">The item to remove from the DoubleLinkedList(T).</param>
/// <returns>True if an item was removed, false otherwise.</returns>
public bool Remove(T item)
{
    return Remove(item, false);
}

/// <summary>
/// Removes the first or all occurrences of the specified item from the DoubleLinkedList(T).
/// </summary>
/// <param name="item">The item to remove from the DoubleLinkedList(T).</param>
/// <param name="allOccurrences">
/// True if all nodes should be removed that contain the specified item, False otherwise
/// </param>
/// <returns>True if an item was removed, false otherwise.</returns>
public bool Remove(T item, bool allOccurrences)
{
    if (IsEmpty)
    {
        return false;
    }

    EqualityComparer<T> comparer = EqualityComparer<T>.Default;
    bool removed = false;
    DoubleLinkedListNode<T> curr = Head;

    while (curr != null)
    {
        // Check to see if the current node contains the data we are trying to delete
        if (!comparer.Equals(curr.Data, item))
        {
            // Assign the current node to the previous node
            // and the previous node to the current node
            curr = curr.Next;
            continue;
        }

        // Create a pointer to the next node in the previous node
        if (curr.Previous != null)

```

```
{  
    curr.Previous.Next = curr.Next;  
}  
  
// Create a pointer to the previous node in the next node  
if (curr.Next != null)  
{  
    curr.Next.Previous = curr.Previous;  
}  
  
if (curr == Head)  
{  
    // If the current node is the head we will have to  
    // assign the next node as the head  
    Head = curr.Next;  
}  
  
if (curr == Tail)  
{  
    // If the current node is the tail we will have to  
    // assign the previous node as the tail  
    Tail = curr.Previous;  
}  
  
// Save the pointer for clean up later  
DoubleLinkedListNode<T> tmp = curr;  
  
// Advance the current to the next node  
curr = curr.Next;  
  
// Since the node will no longer be used clean up the pointers in it  
tmp.Next = null;  
tmp.Previous = null;  
tmp.Owner = null;  
  
// Decrement the counter since we have removed a node  
--m_count;  
removed = true;  
  
if (!allOccurrences)  
{  
    break;  
}  
}  
  
if (removed)  
{  
    ++m_updateCode;  
}  
  
return removed;  
}
```

```
/// <summary>
/// Removes the specified node from the DoubleLinkedList(T).
/// </summary>
/// <param name="node">The node to remove from the DoubleLinkedList(T).</param>
public void Remove(DoubleLinkedListNode<T> node)
{
    if (IsEmpty)
    {
        return;
    }

    if (node == null)
    {
        throw new ArgumentNullException("node");
    }

    if (node.Owner != this)
    {
        throw new InvalidOperationException("The node doesn't belong to this list.");
    }

    DoubleLinkedListNode<T> prev = node.Previous;
    DoubleLinkedListNode<T> next = node.Next;

    // Assign the head to the next node if the specified node is the head
    if (m_head == node)
    {
        m_head = next;
    }

    // Assign the tail to the previous node if the specified node is the tail
    if (m_tail == node)
    {
        m_tail = prev;
    }

    // Set the previous node next reference to the removed nodes next reference.
    if (prev != null)
    {
        prev.Next = next;
    }

    // Set the next node prev reference to the removed nodes prev reference.
    if (next != null)
    {
        next.Previous = prev;
    }

    // Null out the removed nodes next and prev pointer to be safe.
    node.Previous = null;
    node.Next = null;
    node.Owner = null;

    --m_count;
    ++m_updateCode;
}
```

**Visual Basic**

```
''' <summary>
''' Removes the first occurrence of the specified item from the DoubleLinkedList(T).
''' </summary>
''' <param name="item">The item to remove from the DoubleLinkedList(T).</param>
''' <returns>True if an item was removed, false otherwise.</returns>
Public Function Remove(ByVal item As T) As Boolean
    Return Remove(item, False)
End Function

''' <summary>
''' Removes the first or all occurrences of the specified item from the DoubleLinkedList(T).
''' </summary>
''' <param name="item">The item to remove from the DoubleLinkedList(T).</param>
''' <param name="allOccurrences">
''' True if all nodes should be removed that contain the specified item, False otherwise
''' </param>
''' <returns>True if an item was removed, false otherwise.</returns>
Public Function Remove(ByVal item As T, ByVal allOccurrences As Boolean) As Boolean
    If (IsEmpty) Then
        Return False
    End If

    Dim comparer As EqualityComparer(Of T) = EqualityComparer(Of T).Default
    Dim removed As Boolean = False
    Dim curr As DoubleLinkedListNode(Of T) = Head

    While (curr IsNot Nothing)
        ' Check to see if the current node contains the data we are trying to delete
        If (Not comparer.Equals(curr.Data, item)) Then
            ' Assign the current node to the previous node
            ' and the previous node to the current node
            curr = curr.Next
            Continue While
        End If

        ' Create a pointer to the next node in the previous node
        If (curr.Previous IsNot Nothing) Then
            curr.Previous.Next = curr.Next
        End If

        ' Create a pointer to the previous node in the next node
        If (curr.Next IsNot Nothing) Then
            curr.Next.Previous = curr.Previous
        End If

        If (curr Is Head) Then
            ' If the current node is the head we will have to
            ' assign the next node as the head
            Head = curr.Next
        End If
    End While
End Function
```

```
If (curr Is Tail) Then
    ' If the current node is the tail we will have to
    ' assign the previous node as the tail
    Tail = curr.Previous
End If

    ' Save the pointer for clean up later
Dim tmp As DoubleLinkedListNode(Of T) = curr

    ' Advance the current to the next node
curr = curr.Next

    ' Since the node will no longer be used clean up the pointers in it
tmp.Next = Nothing
tmp.Previous = Nothing
tmp.Owner = Nothing

    ' Decrement the counter since we have removed a node
m_count -= 1
removed = True

If (Not allOccurrences) Then
    Exit While
End If
End While

If (removed) Then
    m_updateCode += 1
End If

Return removed
End Function

''' <summary>
''' Removes the specified node from the DoubleLinkedList(T).
''' </summary>
''' <param name="node">The node to remove from the DoubleLinkedList(T).</param>
Public Sub Remove(ByVal node As DoubleLinkedListNode(Of T))
    If (IsEmpty) Then
        Return
    End If

    If (node Is Nothing) Then
        Throw New ArgumentNullException("node")
    End If

    If (node.Owner IsNot Me) Then
        Throw New InvalidOperationException("The node doesn't belong to this list.")
    End If

    Dim prev As DoubleLinkedListNode(Of T) = node.Previous
    Dim pnext As DoubleLinkedListNode(Of T) = node.Next
```

```

' Assign the head to the next node if the specified node is the head
If (m_head Is node) Then
    m_head = pnext
End If

' Assign the tail to the previous node if the specified node is the tail
If (m_tail Is node) Then
    m_tail = prev
End If

' Set the previous node next reference to the removed nodes next reference.
If (prev IsNot Nothing) Then
    prev.Next = pnext
End If

' Set the next node prev reference to the removed nodes prev reference.
If (pnext IsNot Nothing) Then
    pnext.Previous = prev
End If

' nothing out the removed nodes next and prev pointer to be safe.
node.Previous = Nothing
node.Next = Nothing
node.Owner = Nothing

m_count -= 1
m_updateCode += 1
End Sub

```

With the *Remove* method, you can remove an item by using its node or value. If the value is specified, the method traverses the list until it finds the node that contains the value. The implementation for removing it is done as previously described.

With the *Clear* method, users can remove all nodes in the list without having to repeatedly call the *Remove* method with the *Head* pointer until the *Count* is 0.

### C#

```

/// <summary>
/// Removes all items from the DoubleLinkedList(T).
/// </summary>
public void Clear()
{
    DoubleLinkedListNode<T> tmp;

    // Clean up the items in the list
    for (DoubleLinkedListNode<T> node = m_head; node != null; )
    {
        tmp = node.Next;

        // Change the count and head pointer in case we throw an exception.
        // this way the node is removed before we clear the data
    }
}

```

```

m_head = tmp;
if (tmp != null)
{
    tmp.Previous = null;
}
--m_count;

// Erase the contents of the node
node.Next = null;
node.Previous = null;
node.Owner = null;

// Move to the next node
node = tmp;
}

if (m_count <= 0)
{
    m_head = null;
    m_tail = null;
}

++m_updateCode;
}

```

### Visual Basic

```

''' <summary>
''' Removes all items from the DoubleLinkedList(T).
''' </summary>
Public Sub Clear()
    Dim tmp As DoubleLinkedListNode(Of T)

    ' Clean up the items in the list
    Dim node As DoubleLinkedListNode(Of T) = m_head
    While (node IsNot Nothing)
        tmp = node.Next

        ' Change the count and head pointer in case we throw an exception.
        ' this way the node is removed before we clear the data
        m_head = tmp
        If (tmp IsNot Nothing) Then
            tmp.Previous = Nothing
        End If
        m_count -= 1

        ' Erase the contents of the node
        node.Next = Nothing
        node.Previous = Nothing
        node.Owner = Nothing

        ' Move to the next node
        node = tmp
    End While

```

```

If (m_count <= 0) Then
    m_head = Nothing
    m_tail = Nothing
End If

m_updateCode += 1
End Sub

```

## Adding Helper Methods and Properties

Users will want the ability to check the status of the list. The *Contains* and *Find* methods let them look at the contents of the list, whereas the *Head*, *Tail*, *Count*, and *IsEmpty* properties allow them to look at the status of the list.

Your users may find it necessary to find the node of an item in the list or check to see if a node is present in the list. This information can stop them from having to unnecessarily traverse or operate on the list. The following method and properties allow users to find or check for the presence of a node.

The *Find* method allows users to find or check for the presence of a node. It locates a node in the list by searching for the value it contains, as follows.

### C#

```

///<summary>
/// Locates the first node that contains the specified data.
///</summary>
///<param name="data">The data to find.</param>
///<returns>The node that contains the specified data, null otherwise.</returns>
public DoubleLinkedListNode<T> Find(T data)
{
    if (IsEmpty)
    {
        return null;
    }

    EqualityComparer<T> comparer = EqualityComparer<T>.Default;

    // Traverse the list from Head to tail
    for (DoubleLinkedListNode<T> curr = Head; curr != null; curr = curr.Next)
    {
        // Return the node we are currently on if it contains the data we are looking for.
        if (comparer.Equals(curr.Data, data))
        {
            return curr;
        }
    }

    return null;
}

```

### Visual Basic

```
''' <summary>
''' Locates the first node that contains the specified data.
''' </summary>
''' <param name="data">The data to find.</param>
''' <returns>The node that contains the specified data, nothing otherwise.</returns>
Public Function Find(ByVal data As T) As DoubleLinkedListNode(Of T)
    If (IsEmpty) Then
        Return Nothing
    End If

    Dim comparer As EqualityComparer(Of T) = EqualityComparer(Of T).Default

    ' Traverse the list from Head to tail
    Dim curr As DoubleLinkedListNode(Of T) = Head
    While (curr IsNot Nothing)
        ' Return the node we are currently on if it contains the data we are looking for.
        If (comparer.Equals(curr.Data, data)) Then
            Return curr
        End If

        curr = curr.Next
    End While

    Return Nothing
End Function
```

The method traverses the list from the beginning until it finds the node that contains the data you are looking for. A *null* is returned if no node was found.

The *Contains* method is used to check to see if an item is present in the list.

### C#

```
/// <summary>
/// Checks if the specified data is present in the DoubleLinkedList(T).
/// </summary>
/// <param name="data">The data to look for.</param>
/// <returns>True if the data is found, false otherwise.</returns>
public bool Contains(T data)
{
    return Find(data) != null;
}
```

### Visual Basic

```
''' <summary>
''' Checks if the specified data is present in the DoubleLinkedList(T).
''' </summary>
''' <param name="data">The data to look for.</param>
''' <returns>True if the data is found, false otherwise.</returns>
Public Function Contains(ByVal data As T) As Boolean
    Return Find(data) IsNot Nothing
End Function
```

The method works by checking to see if the *Find* method found the specified node.

Users may also need to traverse the list for other reasons as well. To do this, they will need access to the head and tail.

### C#

```
/// <summary>
/// Gets the head node of the DoubleLinkedList(T).
/// </summary>
public DoubleLinkedListNode<T> Head
{
    get { return m_head; }
    private set { m_head = value; }
}

/// <summary>
/// Gets the tail node of the DoubleLinkedList(T).
/// </summary>
public DoubleLinkedListNode<T> Tail
{
    get { return m_tail; }
    private set { m_tail = value; }
}
```

### Visual Basic

```
'<summary>
'Gets the head node of the DoubleLinkedList(T).
'</summary>
Public Property Head() As DoubleLinkedListNode(Of T)
    Get
        Return m_head
    End Get
    Private Set(ByVal value As DoubleLinkedListNode(Of T))
        m_head = value
    End Set
End Property

'<summary>
'Gets the tail node of the DoubleLinkedList(T).
'</summary>
Public Property Tail() As DoubleLinkedListNode(Of T)
    Get
        Return m_tail
    End Get
    Private Set(ByVal value As DoubleLinkedListNode(Of T))
        m_tail = value
    End Set
End Property
```

The *Head* and *Tail* methods return the head and tail of the list respectively. Both properties block the setting of the values by the user.

Knowing how many items are in the list and if it is empty eliminates the need to do some operations on the list. The following properties help users determine what needs to be done.

### C#

```
/// <summary>
/// States if the DoubleLinkedList(T) is empty.
/// </summary>
public bool IsEmpty
{
    get { return m_count <= 0; }
}

/// <summary>
/// Gets the number of elements actually contained in the DoubleLinkedList(T).
/// </summary>
public int Count
{
    get { return m_count; }
}
```

### Visual Basic

```
''' <summary>
''' States if the DoubleLinkedList(T) is empty.
''' </summary>
Public ReadOnly Property IsEmpty() As Boolean
    Get
        Return m_count <= 0
    End Get
End Property

''' <summary>
''' Gets the number of elements actually contained in the DoubleLinkedList(T).
''' </summary>
Public ReadOnly Property Count() As Integer
    Get
        Return m_count
    End Get
End Property
```

The following helper function helps copy the contents of the linked list into a new array. The new array is useful when code requires an array instead of a *DoubleLinked(T)* class.

### C#

```
/// <summary>
/// Copies the elements of the DoubleLinkedList(T) to a new array.
/// </summary>
/// <returns>
/// An array containing copies of the elements of the DoubleLinkedList(T).
/// </returns>
```

```
public T[] ToArray()
{
    T[] retval = new T[m_count];

    int index = 0;
    for (DoubleLinkedListNode<T> i = Head; i != null; i = i.Next)
    {
        retval[index] = i.Data;
        ++index;
    }

    return retval;
}
```

**Visual Basic**

```
''' <summary>
''' Copies the elements of the DoubleLinkedList(T) to a new array.
''' </summary>
''' <returns>
''' An array containing copies of the elements of the DoubleLinkedList(T).
''' </returns>
Public Function ToArray() As T()
    Dim retval As T() = New T(m_count - 1) {}

    Dim index As Integer = 0
    Dim i As DoubleLinkedListNode(Of T) = Head
    While (i IsNot Nothing)
        retval(index) = i.Data
        index += 1
        i = i.Next
    End While

    Return retval
End Function
```

Because the list is a doubly linked list, the same amount of time is required to traverse the list from head to tail and from tail to head. The following method puts the contents in an array in reverse order.

**C#**

```
/// <summary>
/// Copies the elements of the DoubleLinkedList(T) from back to front to a new array.
/// </summary>
/// <returns>
/// An array containing copies of the elements of the DoubleLinkedList<T>.
/// </returns>
public T[] ToArrayReversed()
{
    T[] retval = new T[m_count];

    int index = 0;
    for (DoubleLinkedListNode<T> i = Tail; i != null; i = i.Previous)
    {
        retval[index] = i.Data;
```

```
    ++index;  
}  
  
return retval;  
}
```

### Visual Basic

```
''' <summary>  
''' Copies the elements of the DoubleLinkedList(T) from back to front to a new array.  
''' </summary>  
''' <returns>  
''' An array containing copies of the elements of the DoubleLinkedList(T).  
''' </returns>  
Public Function ToArrayReversed() As T()  
    Dim retval As T() = New T(m_count - 1) {}  
  
    Dim index As Integer = 0  
    Dim i As DoubleLinkedListNode(Of T) = Tail  
    While (i IsNot Nothing)  
        retval(index) = i.Data  
        index += 1  
        i = i.Previous  
    End While  
  
    Return retval  
End Function
```

## Using an Array to Create a Linked List

Instead of creating nodes when they are needed and having them store object references, you could implement the linked list as an array. Using an array as a linked list will be discussed in Chapter 2, “Understanding Collections: Associative Arrays.”

## Using the Linked List Class

Consider this scenario to practice using the linked list class. Your boss decides that the requirements of your last exercise should change. Rather than sort an array of 20 numbers, he wants you to sort 10 numbers as you get them. You realize that because this will require a lot of insertions, arrays are not the greatest choice. You remember the chapter you read on linked lists and decide to use a doubly linked list to hold the sorted values.

### C#

```
using DevGuideToCollections;  
using System.Text;
```

### Visual Basic

```
Imports DevGuideToCollections  
Imports System.Text
```

For C#, in the *Program* class, create a method called *Lesson2A* as follows.

#### C#

```
static void Lesson2A()
{
}
```

For Visual Basic, in the *Module1* module, create a method called *Lesson2a* as follows.

#### Visual Basic

```
Sub Lesson2A()
End Sub
```

Create the following helper method.

#### C#

```
static string ArrayToString(Array array)
{
    StringBuilder sb = new StringBuilder();

    sb.Append("[");
    if (array.Length > 0)
    {
        sb.Append(array.GetValue(0));
    }
    for (int i = 1; i < array.Length; ++i)
    {
        sb.AppendFormat(",{0}", array.GetValue(i));
    }
    sb.Append("]");

    return sb.ToString();
}
```

#### Visual Basic

```
Function ArrayToString(ByVal array As Array) As String
    Dim sb As StringBuilder = New StringBuilder()

    sb.Append("[")
    If (array.Length > 0) Then
        sb.Append(array.GetValue(0))
    End If
    For i As Integer = 1 To array.Length - 1
        sb.AppendFormat(",{0}", array.GetValue(i))
    Next
    sb.Append("]")

    Return sb.ToString()
End Function
```

This helper method converts the elements of an array to a string. In Chapter 6, you will learn how to traverse the *DoubleLinkedList(T)* with a *foreach* statement.

The driver for method *Lesson2a* looks as follows.

### C#

```
Random rnd = new Random();
DoubleLinkedList<int> list = new DoubleLinkedList<int>();

Console.WriteLine("Adding to the list...");

for (int i = 0; i < 10; ++i)
{
    // Get the value to add
    int nextValue = rnd.Next(100);

    Console.Write("{0} ", nextValue);

    bool added = false;

    . . .

}

Console.WriteLine();

Console.WriteLine("The sorted list is");
Console.WriteLine(ArrayToString(list.ToArray()));
```

### Visual Basic

```
Dim rnd As Random = New Random()
Dim list As DoubleLinkedList(Of Integer) = New DoubleLinkedList(Of Integer)()

Console.WriteLine("Adding to the list...")

For i As Integer = 0 To 9
    ' Get the value to add
    Dim nextValue As Integer = rnd.Next(100)

    Console.Write("{0} ", nextValue)

    Dim added As Boolean = False

    . . .

Next

Console.WriteLine()

Console.WriteLine("The sorted list is")
Console.WriteLine(ArrayToString(list.ToArray()))
```

The preceding driver creates 10 random numbers. To sort the numbers as you get them, you can traverse your sorted list until you find a value that is greater than the number you are adding to the list. After you have added the item to the list, you should set a flag that states that the item is added to the list, as follows.

**C#**

```
// Traverse the list until you find the item greater than nextValue.
for (DoubleLinkedListNode<int> curr = list.Head; curr != null; curr = curr.Next)
{
    // If the item is less than the current value, you need to insert item before the
    // current node.
    if (nextValue < curr.Data)
    {
        list.AddBefore(curr, nextValue);

        // Mark the item as added
        added = true;

        // Exit the loop
        break;
    }
}
```

**Visual Basic**

```
' Traverse the list until you find the item greater than nextValue.
Dim curr As DoubleLinkedListNode(Of Integer) = list.Head
While (curr IsNot Nothing)

    ' If the item is less than the current value, you need to insert item before the
    ' current node.
    If (nextValue < curr.Data) Then
        list.AddBefore(curr, nextValue)

        ' Mark the item as added
        added = True

        Exit While
    End If

    curr = curr.Next
End While
```

If the item hasn't been added to the list, you need to add it to the end of the list by checking the added flag as follows.

**C#**

```
// If the item has not been added to the list, the item is either greater than
// all items in the list or the list is empty. In either case, the item should be
// added to the end of the list.
if (!added)
{
    list.AddToEnd(nextValue);
}
```

### Visual Basic

```
' If the item has not been added to the list, the item is either greater than
' all items in the list or the list is empty. In either case, the item should be
' added to the end of the list.
If (Not added) Then
    list.AddToEnd(nextValue)
End If
```

The executed code will resemble the following.

### Output

```
Adding to the list...
10 5 40 5 29 41 35 33 29 94
The sorted list is
[5,5,10,29,29,33,35,40,41,94]
```

The executed code will display the items being sorted and then the sorted list.

## Summary

In this chapter, you began your journey through collections by learning about arrays and linked lists. You saw how arrays are accessed through indexes and how linked lists are accessed through nodes. You also learned of the advantages and disadvantages of both and when to use one instead of the other.



## Chapter 2

# Understanding Collections: Associative Arrays

After completing this chapter, you will be able to

- Identify associative arrays.
- Design and implement associative arrays.
- Understand when and when not to use associative arrays.

## Associative Array Overview

An *associative array*, also called a *dictionary*, is an abstract collection type that associates a single key with a value. Each key in the collection is unique. Some languages refer to an associative array as a *hash table*, which generally leads to some confusion regarding the difference between an associative array and a hash table. An associative array defines how a collection should behave, such as associating a key with a value. You can implement an associative array in many ways. One possible way is by using a hash table, which is why some languages refer to an associative array as a hash table.

## Uses of Associative Arrays

Associative arrays are useful when you have a value you need to look up with a unique key. You should use other collection types if you never need to use the key to find the value. Say you have a *Person* class that contains personnel information that you need to store and access quickly whenever someone gives you a phone number. If the collection contains hundreds of instances of the *Person* class, you would have to constantly traverse the collection to locate the specific record. This would be a performance hit if you had to search thousands or even hundreds of records every second. An associative array that uses a good lookup implementation would be able to access the *Person* instance quickly through his or her phone number.

## Advantages and Disadvantages of Associative Arrays

The advantages and disadvantages of an associative array are normally determined by the implementation you use for the key/value association. Some implementations, such as association lists, are very easy to implement but very inefficient for large lists. Others, such as red-black trees and hash tables, are harder to implement but a lot more efficient for larger lists, and not so efficient for smaller lists. If you are willing to do the research and invest the coding time, associative arrays can be a very efficient tool.

## Associative Array Implementation

You can implement an associative array in multiple ways. Each way has its advantages and disadvantages. Two of the most common ways are red-black trees and hash tables. In the following sections, you create an associative array using an association list and a hash table.



**More Info** Other methods for creating associative arrays, such as trees and skip lists, are beyond the scope of this book. You can find information about those methods in other books or on the Internet.

### Using Association Lists for Associative Arrays

An association list uses a linked list to associate keys to values. Each node in the linked list stores a key/value pair. When the user requests a value with a key, the list is searched from the head to the tail to locate the node that contains the key.

#### Advantages of Association Lists

Association lists are very easy to implement and debug. They can also outperform other types of associative array implementation when the size of the list is small. Adding to the list isn't as complex as with other types of associative array implementations.

#### Disadvantages of Association Lists

The list has to be traversed from the beginning of the list to the end to locate a key. So, an association list will perform worse on larger lists than some other associative array implementations.

## Getting Started

You can create an associative array class called *AssociativeArrayAL(T)* by using an association list, as demonstrated in the following code.

C#

```
[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(AssociativeArrayDebugView))]
public class AssociativeArrayAL<TKey, TValue>
{
    // Fields
    private IEqualityComparer<TKey> m_comparer;
    private DoubleLinkedList<KeyValuePair<TKey, TValue>> m_list;
    private int m_updateCode;

    // Constructor
    public AssociativeArrayAL();
    public AssociativeArrayAL(IEqualityComparer<TKey> comparer);

    // Methods
    public void Add(TKey key, TValue value);
    private void Add(TKey key, TValue value, bool overwrite);
    public void Clear();
    public bool ContainsKey(TKey key);
    public bool ContainsValue(TValue value);
    private DoubleListNode<KeyValuePair<TKey, TValue>> FindKey(TKey key);
    private DoubleListNode<KeyValuePair<TKey, TValue>> FindValue(TValue value);
    public bool Remove(TKey key);
    private bool Remove(DoubleListNode<KeyValuePair<TKey, TValue>> node);
    public bool RemoveValue(TValue value);
    public bool RemoveValue(TValue value, bool allOccurrences);
    public bool TryGetValue(TKey key, out TValue value);

    // Properties
    public int Count { get; }
    public bool IsEmpty { get; }
    public TValue this[TKey key] { get; set; }
    public TKey[] Keys { get; }
    public TValue[] Values { get; }

    // Nested Types
    private struct KeyValuePair
    {
        private TKey m_key;
        private TValue m_value;
        public KeyValuePair(TKey key, TValue value);
        public TKey Key { get; }
        public TValue Value { get; set; }
    }
}
```

**Visual Basic**

```

<DefaultMember("Item")> _
<DebuggerDisplay("Count={Count}")> _
<DebuggerTypeProxy(GetType(AssociativeArrayDebugView))> _
Public Class AssociativeArrayAL(Of TKey, TValue)

    ' Fields
    Private m_comparer As IEqualityComparer(Of TKey)
    Private m_list As DoubleLinkedList(Of KeyValuePair(Of TKey, TValue))
    Private m_updateCode As Integer

    ' Constructors
    Public Sub New()
        Public Sub New(ByVal comparer As IEqualityComparer(Of TKey))

    ' Methods
    Public Sub Add(ByVal key As TKey, ByVal value As TValue)
    Private Sub Add(ByVal key As TKey, ByVal value As TValue, ByVal overwrite As Boolean)
    Public Sub Clear()
    Public Function ContainsKey(ByVal key As TKey) As Boolean
    Public Function ContainsValue(ByVal value As TValue) As Boolean
    Private Function FindKey(ByVal key As TKey) _
        As DoubleLinkedListNode(Of KeyValuePair(Of TKey, TValue))
    Private Function FindValue(ByVal value As TValue) _
        As DoubleLinkedListNode(Of KeyValuePair(Of TKey, TValue))
    Public Function Remove(ByVal key As TKey) As Boolean
    Private Function Remove(ByVal node As DoubleLinkedListNode(Of KeyValuePair(Of TKey, TValue))) _
        As Boolean
    Public Function RemoveValue(ByVal value As TValue) As Boolean
    Public Function RemoveValue(ByVal value As TValue, ByVal allOccurrences As Boolean) _
        As Boolean
    Public Function TryGetValue(ByVal key As TKey, ByRef value As TValue) As Boolean

    ' Properties
    Public ReadOnly Property Count As Integer
    Public ReadOnly Property IsEmpty As Boolean
    Public Default Property Item(ByVal key As TKey) As TValue
    Public ReadOnly Property Keys As TKey()
    Public ReadOnly Property Values As TValue()

    ' Nested Types
    <StructLayout(LayoutKind.Sequential)> _
    Private Structure KeyValuePair
        Public m_key As TKey
        Public m_value As TValue
        Public Sub New(ByVal key As TKey, ByVal value As TValue)
        Public ReadOnly Property Key As TKey
        Public Property Value As TValue
    End Structure
End Class

```

The internal storage will be implemented as a doubly linked list stored in the field *m\_list*. The comparer that is used for comparing keys will be stored in *m\_comparer*. The *m\_comparer* field will be used to check whether a key matches a key stored in *m\_list*.

The *m\_updateCode* field will be incremented each time the user modifies the list. The *m\_updateCode* field will be used in Chapter 6, ".NET Collection Interfaces," to determine whether the collection has changed while the user is iterating over it. It is easier to add it to the code now instead of changing the code in Chapter 6.

An association list works by storing the key/value pair in a linked list. To accomplish this, you need a data type to store the key and value. The *KVPair* struct will be used to store the key and value of each association.

## Creating Constructors

The *AssociativeArrayAL(T)* class will contain two constructors. One constructor is for creating an empty class, and the other is for specifying a comparer to be used for the key comparisons.

With the default constructor for the *AssociativeArrayAL(T)* class, users can create an empty associative array.

### C#

```
/// <summary>
/// Initializes a new instance of the AssociativeArrayAL(TKey, TValue) class
/// that is empty.
/// </summary>
public AssociativeArrayAL()
{
    m_comparer = EqualityComparer<TKey>.Default;
    m_list = new DoubleLinkedList<KeyValuePair>();
}
```

### Visual Basic

```
''' <summary>
''' Initializes a new instance of the AssociativeArrayAL(TKey, TValue) class
''' that is empty.
''' </summary>
Public Sub New()
    m_comparer = EqualityComparer(Of TKey).Default
    m_list = New DoubleLinkedList(Of KeyValuePair)()
End Sub
```

With the next constructor, users can specify the comparer that will be used to do key comparisons.

### C#

```
/// <summary>
/// Initializes a new instance of the AssociativeArrayAL(TKey, TValue) class
/// that is empty and uses the specified comparer.
/// </summary>
/// <param name="comparer">The comparer to use for the keys.</param>
public AssociativeArrayAL(IEqualityComparer<TKey> comparer)
{
```

```

    if (comparer == null)
    {
        throw new ArgumentNullException("comparer");
    }

    m_comparer = comparer;
    m_list = new DoubleLinkedList<KeyValuePair>();
}

```

### Visual Basic

```

''' <summary>
''' Initializes a new instance of the AssociativeArrayAL(TKey, TValue) class
''' that is empty and uses the specified comparer.
''' </summary>
''' <param name="comparer">The comparer to use for the keys.</param>
Public Sub New(ByVal comparer As IEqualityComparer(Of TKey))
    If (comparer Is Nothing) Then
        Throw New ArgumentNullException("comparer")
    End If

    m_comparer = comparer
    m_list = New DoubleLinkedList(Of KeyValuePair)()
End Sub

```

## Allowing Users to Associate Values with Keys

Users need the ability to associate values with keys and to reassign keys as well. They can accomplish this using the *Add* method and *Item* property. The *Add* method and *Item* property are defined as follows.

### C#

```

/// <summary>
/// Adds the key value pair to the AssociativeArrayAL(TKey, TValue).
/// </summary>
/// <param name="key">The key to associate with the value.</param>
/// <param name="value">The value to add.</param>
public void Add(TKey key, TValue value)
{
    Add(key, value, false);
}

/// <summary>
/// Gets or sets the value at the specified key.
/// </summary>
/// <param name="key">The key to use for finding the value.</param>
/// <returns>The value associated with the specified key.</returns>
public TValue this[TKey key]
{
    set
    {
        Add(key, value, true);
    }
}

```

```
    }  
}
```

### Visual Basic

```
''' <summary>  
''' Adds the key value pair to the AssociativeArrayAL(TKey,TValue).  
''' </summary>  
''' <param name="key">The key to associate with the value.</param>  
''' <param name="value">The value to add.</param>  
Public Sub Add(ByVal key As TKey, ByVal value As TValue)  
    Add(key, value, False)  
End Sub  
  
''' <summary>  
''' Gets or sets the value at the specified key.  
''' </summary>  
''' <param name="key">The key to use for finding the value.</param>  
''' <returns>The value associated with the specified key.</returns>  
Default Public Property Item(ByVal key As TKey) As TValue  
    set  
        Add(key, Value, True)  
    End Set  
End Property
```

Both call the *Add(TKey,TValue,bool)* method that is defined as follows.

### C#

```
/// <summary>  
/// Adds the key value pair to the AssociativeArrayAL(TKey,TValue)  
/// </summary>  
/// <param name="key">The key to associate with the specified value.</param>  
/// <param name="value">The value to add to the AssociativeArrayAL(TKey,TValue).</param>  
/// <param name="overwrite">  
/// True if the value should be overwritten if it exist, false if an error should be thrown.  
/// </param>  
void Add(TKey key, TValue value, bool overwrite)  
{  
    DoubleLinkedListNode<KeyValuePair> node = FindKey(key);  
    if (node != null)  
    {  
        if (!overwrite)  
        {  
            throw new InvalidOperationException("The specified key is already present");  
        }  
        else  
        {  
            KeyValuePair tmp = node.Data;  
            tmp.Value = value;  
            node.Data = tmp;  
        }  
    }  
}
```

```

        return;
    }

    KVPair kvp = new KVPair(key, value);

    m_list.AddToBeginning(kvp);

    ++m_updateCode;
}

```

### Visual Basic

```

''' <summary>
''' Adds the key value pair to the AssociativeArrayAL(TKey,TValue)
''' </summary>
''' <param name="key">The key to associate with the specified value.</param>
''' <param name="value">The value to add to the AssociativeArrayAL(TKey,TValue).</param>
''' <param name="overwrite">
''' True if the value should be overwritten if it exist, false if an error should be thrown.
''' </param>
Sub Add(ByVal key As TKey, ByVal value As TValue, ByVal overwrite As Boolean)
    Dim node As DoubleLinkedListNode(Of KVPair) = FindKey(key)
    If (node IsNot Nothing) Then
        If (Not overwrite) Then
            Throw New InvalidOperationException("The specified key is already present")
        Else
            Dim tmp As KVPair = node.Data
            tmp.Value = value
            node.Data = tmp
        End If
        Return
    End If

    Dim kvp As KVPair = New KVPair(key, value)

    m_list.AddToBeginning(kvp)

    m_updateCode += 1
End Sub

```

The *Add(TKey,TValue,bool)* method searches through the internal linked list to see whether the key is already present in the list. If the key is already present, an *InvalidOperationException* will be thrown if the user wasn't trying to reassign the key. If the key exists and the user was trying to reassign the key, the method will reassign the key and return. If the key doesn't exist, the method will add the key/value pair to the beginning of the list and increment the update code.

## Allowing Users to Remove Keys

Keys also need to be removed when they are no longer needed. The *Clear*, *Remove*, and *RemoveValue* methods are used to do this.

The *Clear* method removes all key/value pairs from the collection and is defined as follows.

### C#

```
/// <summary>
/// Removes all items from the AssociativeArrayAL(TKey, TValue).
/// </summary>
public void Clear()
{
    m_list.Clear();
    ++m_updateCode;
}
```

### Visual Basic

```
'> <summary>
'> Removes all items from the AssociativeArrayAL(TKey, TValue).
'> </summary>
Public Sub Clear()
    m_list.Clear()
    m_updateCode += 1
End Sub
```

The *Remove* method removes the key/value pair associated with the specified key and is defined as follows.

### C#

```
/// <summary>
/// Removes the specified key from the AssociativeArrayAL(TKey, TValue).
/// </summary>
/// <param name="key">The key to remove from the AssociativeArrayAL(TKey, TValue).</param>
/// <returns>True if the key was removed, false otherwise.</returns>
public bool Remove(TKey key)
{
    DoubleLinkedListNode<KeyValuePair> node = FindKey(key);

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

    return Remove(node);
}
```

### Visual Basic

```
''<summary>
''' Removes the specified key from the AssociativeArrayAL(TKey,TValue).
''' </summary>
''' <param name="key">The key to remove from the AssociativeArrayAL(TKey,TValue).</param>
''' <returns>True if the key was removed, false otherwise.</returns>
Public Function Remove(ByVal key As TKey) As Boolean
    Dim node As DoubleLinkedListNode(Of KVPpair) = FindKey(key)

    If (node Is Nothing) Then
        Return False
    End If

    Return Remove(node)
End Function
```

The *Remove(DoubleLinkedListNode(KVPpair))* and *FindKey* methods are discussed later in this chapter.

The *RemoveValue* method removes the key/value pair that contains the specified value and is defined as follows.

### C#

```
///<summary>
/// Removes the first occurrence of the specified value.
/// </summary>
/// <param name="value">The value to remove.</param>
/// <returns>
/// True if the value was removed,
/// false if it wasn't present in the AssociativeArrayAL(TKey,TValue).
/// </returns>
public bool RemoveValue(TValue value)
{
    return RemoveValue(value, false);
}

///<summary>
/// Removes the specified value.
/// </summary>
/// <param name="value">The value to remove.</param>
/// <param name="allOccurrences">
/// True if all occurrences of the value should be removed, false if not.
/// </param>
/// <returns>
/// True if the value was removed,
/// false if it wasn't present in the AssociativeArrayAL(TKey,TValue).
/// </returns>
public bool RemoveValue(TValue value, bool allOccurrences)
{
    bool removed = false;

    DoubleLinkedListNode<KVPpair> node = FindValue(value);
    while (node != null)
```

```
{  
    removed = Remove(node) || removed;  
  
    if (!allOccurrences)  
    {  
        return removed;  
    }  
  
    node = FindValue(value);  
}  
  
return removed;  
}
```

### Visual Basic

```
''' <summary>  
''' Removes the first occurrence of the specified value.  
''' </summary>  
''' <param name="value">The value to remove.</param>  
''' <returns>  
''' True if the value was removed,  
''' false if it wasn't present in the AssociativeArrayAL(TKey,TValue).  
''' </returns>  
Public Function RemoveValue(ByVal value As TValue) As Boolean  
    Return RemoveValue(value, False)  
End Function  
  
''' <summary>  
''' Removes the specified value.  
''' </summary>  
''' <param name="value">The value to remove.</param>  
''' <param name="allOccurrences">  
''' True if all occurrences of the value should be removed, false if not.  
''' </param>  
''' <returns>  
''' True if the value was removed,  
''' false if it wasn't present in the AssociativeArrayAL(TKey,TValue).  
''' </returns>  
Public Function RemoveValue(ByVal value As TValue, ByVal allOccurrences As Boolean) _  
    As Boolean  
    Dim removed As Boolean = False  
  
    Dim node As DoubleLinkedListNode(Of KVPair) = FindValue(value)  
    While (node IsNot Nothing)  
        removed = Remove(node) Or removed  
  
        If (Not allOccurrences) Then  
            Return removed  
        End If  
  
        node = FindValue(value)  
    End While  
  
    Return removed  
End Function
```

The *Remove* and *RemoveAll* methods both use the *Remove(DoubleLinkedListNode(KVPair))* method, which is defined as follows.

#### C#

```
bool Remove(DoubleLinkedListNode<KVPair> node)
{
    if (node == null)
    {
        return false;
    }

    m_list.Remove(node);

    ++m_updateCode;

    return true;
}
```

#### Visual Basic

```
Private Function Remove(ByVal node As DoubleLinkedListNode(Of KVPair)) As Boolean
    If (node Is Nothing) Then
        Return False
    End If

    m_list.Remove(node)

    m_updateCode += 1

    Return True
End Function
```

## Adding Helper Methods and Properties

Users can get the number of items in the collection by using the *Count* property.

#### C#

```
/// <summary>
/// Gets the number of items in the AssociativeArrayAL(TKey, TValue).
/// </summary>
public int Count
{
    get { return m_list.Count; }
}
```

#### Visual Basic

```
''' <summary>
''' Gets the number of items in the AssociativeArrayAL(TKey, TValue).
''' </summary>
Public ReadOnly Property Count() As Integer
    Get
        Return m_list.Count
    End Get
End Property
```

With the *IsEmpty* property, users can check whether the collection is empty before doing some operations.

**C#**

```
/// <summary>
/// States if the AssociativeArrayAL(TKey,TValue) is empty.
/// </summary>
public bool IsEmpty
{
    get { return Count <= 0; }
}
```

**Visual Basic**

```
'<summary>
'<summary> States if the AssociativeArrayAL(TKey,TValue) is empty.
'</summary>
Public ReadOnly Property IsEmpty() As Boolean
    Get
        Return Count <= 0
    End Get
End Property
```

Users can also get the list of keys and values in the collection by calling the *Keys* and *Values* properties respectively.

**C#**

```
/// <summary>
/// Gets an array of current keys.
/// </summary>
public TKey[] Keys
{
    get
    {
        int index = 0;
        TKey[] keys = new TKey[Count];

        for (DoubleLinkedListNode<KeyValuePair> curr = m_list.Head; curr != null;
            curr = curr.Next)
        {
            keys[index++] = curr.Data.Key;
        }

        return keys;
    }
}

/// <summary>
/// Gets an array of current values.
/// </summary>
public TValue[] Values
{
    get
    {
```

```

TValue[] values = new TValue[Count];
int index = 0;

for (DoubleLinkedListNode<KeyValuePair> curr = m_list.Head; curr != null;
    curr = curr.Next)
{
    values[index++] = curr.Data.Value;
}

return values;
}
}

```

### Visual Basic

```

''' <summary>
''' Gets an array of current keys.
''' </summary>
Public ReadOnly Property Keys() As TKey()
    Get
        Dim index As Integer = 0
        Dim rkeys As TKey() = New TKey(Count - 1) {}

        Dim curr As DoubleLinkedListNode(Of KeyValuePair) = m_list.Head
        While (curr IsNot Nothing)
            rkeys(index) = curr.Data.Key
            index += 1
            curr = curr.Next
        End While

        Return rkeys
    End Get
End Property

''' <summary>
''' Gets an array of current values.
''' </summary>
Public ReadOnly Property Values() As TValue()
    Get
        Dim rvalues As TValue() = New TValue(Count - 1) {}
        Dim index As Integer = 0

        Dim curr As DoubleLinkedListNode(Of KeyValuePair) = m_list.Head
        While (curr IsNot Nothing)
            rvalues(index) = curr.Data.Value
            index += 1
            curr = curr.Next
        End While

        Return rvalues
    End Get
End Property

```

Both properties traverse the list from the head to the tail and add the values to an array as they traverse. This provides the users with a snapshot of the current *Keys* and *Values*. For now, the *Keys* and *Values* properties are just placeholders. In Chapter 6, you learn how to return an enumerator for the *Keys* and *Values* properties that will stay in sync with the collection.

Sometimes users may want to know if a particular key or value is present in the collection. The *ContainsKey* and *ContainsValue* methods can be used to do so.

### C#

```
/// <summary>
/// Checks to see if the AssociativeArrayAL(TKey,TValue) contains the specified value.
/// </summary>
/// <param name="value">The value to look for.</param>
/// <returns>True if the value was found, false otherwise.</returns>
public bool ContainsValue(TValue value)
{
    return FindValue(value) != null;
}

/// <summary>
/// Checks to see if the specified key is present in the AssociativeArrayAL(TKey,TValue).
/// </summary>
/// <param name="key">The key to look for.</param>
/// <returns>True if the key was found, false otherwise.</returns>
public bool ContainsKey(TKey key)
{
    return FindKey(key) != null;
}
```

### Visual Basic

```
''' <summary>
''' Checks to see if the specified key is present in the AssociativeArrayAL(TKey,TValue).
''' </summary>
''' <param name="key">The key to look for.</param>
''' <returns>True if the key was found, false otherwise.</returns>
Public Function ContainsKey(ByVal key As TKey) As Boolean
    Return FindKey(key) IsNot Nothing
End Function

''' <summary>
''' Checks to see if the AssociativeArrayAL(TKey,TValue) contains the specified value.
''' </summary>
''' <param name="value">The value to look for.</param>
''' <returns>True if the value was found, false otherwise.</returns>
Public Function ContainsValue(ByVal value As TValue) As Boolean
    Return FindValue(value) IsNot Nothing
End Function
```

The *ContainsValue* and *ContainsKey* methods use the *FindValue* and *FindKey* methods to tell whether an item is in the collection. They are defined as follows.

**C#**

```
/// <summary>
/// Finds the node that contains the specified key.
/// </summary>
/// <param name="key">The key to look for.</param>
/// <returns>The node that contains the specified key, otherwise null.</returns>
DoubleLinkedListNode<KeyValuePair> FindKey(TKey key)
{
    if (IsEmpty)
    {
        return null;
    }

    for (DoubleLinkedListNode<KeyValuePair> node = m_list.Head; node != null; node = node.Next)
    {
        if (m_comparer.Equals(node.Data.Key, key))
        {
            return node;
        }
    }

    return null;
}

/// <summary>
/// Finds the node that contains the specified value.
/// </summary>
/// <param name="value">The value to look for.</param>
/// <returns>The first node that contains the specified value, otherwise null.</returns>
DoubleLinkedListNode<KeyValuePair> FindValue(TValue value)
{
    if (IsEmpty)
    {
        return null;
    }

    EqualityComparer<TValue> comparer = EqualityComparer<TValue>.Default;

    for (DoubleLinkedListNode<KeyValuePair> node = m_list.Head; node != null; node = node.Next)
    {
        if (comparer.Equals(node.Data.Value, value))
        {
            return node;
        }
    }

    return null;
}
```

### Visual Basic

```
''' <summary>
''' Finds the node that contains the specified key.
''' </summary>
''' <param name="key">The key to look for.</param>
''' <returns>The node that contains the specified key, otherwise null.</returns>
Private Function FindKey(ByVal key As TKey) As DoubleLinkedListNode(Of KeyValuePair)
    If (IsEmpty) Then
        Return Nothing
    End If

    Dim node As DoubleLinkedListNode(Of KeyValuePair) = m_list.Head
    While (node IsNot Nothing)
        If (m_comparer.Equals(node.Data.Key, key)) Then

            Return node
        End If
        node = node.Next
    End While

    Return Nothing
End Function

''' <summary>
''' Finds the node that contains the specified value.
''' </summary>
''' <param name="value">The value to look for.</param>
''' <returns>The first node that contains the specified value, otherwise null.</returns>
Private Function FindValue(ByVal value As TValue) As DoubleLinkedListNode(Of KeyValuePair)
    If (IsEmpty) Then
        Return Nothing
    End If
    Dim comparer As EqualityComparer(Of TValue) = EqualityComparer(Of TValue).Default

    Dim node As DoubleLinkedListNode(Of KeyValuePair) = m_list.Head
    While (node IsNot Nothing)
        If (comparer.Equals(node.Data.Value, value)) Then
            Return node
        End If
        node = node.Next
    End While

    Return Nothing
End Function
```

Both methods traverse the collection from the head to the tail. The *FindValue* and *FindKey* methods check each node by using a comparer to see whether the specified key or value is present at the current node.

Users need the ability to retrieve a value from the collection by using a key. The *Item* property allows them to do so.

**C#**

```

/// <summary>
/// Gets the value at the specified key.
/// </summary>
/// <param name="key">The key to use for finding the value.</param>
/// <returns>The value associated with the specified key.</returns>
public TValue this[TKey key]
{
    get
    {
        DoubleLinkedListNode<KeyValuePair> node = FindKey(key);

        if (node == null)
        {
            throw new KeyNotFoundException("The specified key couldn't be located");
        }

        return node.Data.Value;
    }
}

```

**Visual Basic**

```

''' <summary>
''' Gets or sets the value at the specified key.
''' </summary>
''' <param name="key">The key to use for finding the value.</param>
''' <returns>The value associated with the specified key.</returns>
Default Public Property Item(ByVal key As TKey) As TValue
    Get
        Dim node As DoubleLinkedListNode(Of KeyValuePair) = FindKey(key)

        If (node Is Nothing) Then
            Throw New KeyNotFoundException("The specified key couldn't be located")
        End If

        Return node.Data.Value
    End Get
End Property

```

The *Item* property uses the *FindKey* method to locate the node that contains the specified key. If the node doesn't exist, the *Item* property throws an exception. This may not always be what the user wants. The *TryGetValue* method allows a user to try and retrieve a value without throwing an exception if the key doesn't exist in the collection.

**C#**

```

/// <summary>
/// Tries to get the value at the specified key without throwing an exception.
/// </summary>
/// <param name="key">The key to get the value for.</param>
/// <param name="value">
/// The value that is associated with the specified key or the default value for the type.
/// </param>
/// <returns>True if the key was found, false otherwise.</returns>

```

```
public bool TryGetValue(TKey key, out TValue value)
{
    DoubleLinkedListNode<KeyValuePair> node = FindKey(key);

    if (node != null)
    {
        value = node.Data.Value;
        return true;
    }

    value = default(TValue);
    return false;
}
```

### Visual Basic

```
''' <summary>
''' Tries to get the value at the specified key without throwing an exception.
''' </summary>
''' <param name="key">The key to get the value for.</param>
''' <param name="value">
'''   The value that is associated with the specified key or the default value for the type.
''' </param>
''' <returns>True if the key was found, false otherwise.</returns>
Public Function TryGetValue(ByVal key As TKey, ByRef value As TValue) As Boolean
    Dim node As DoubleLinkedListNode(Of KeyValuePair) = FindKey(key)

    If (node IsNot Nothing) Then
        value = node.Data.Value
        Return True
    End If

    value = CType(Nothing, TValue)
    Return False
End Function
```

## Using Hash Tables for Associative Arrays

A hash table is a collection type that maps values to keys. All values can be accessed using their keys but not vice versa. All mappings are done using a hash function.

### Advantages of Hash Tables

The hashing function makes hash tables more efficient than searching a collection for a value. With a good hashing function, the amount of time it takes to access any element in a larger collection or a smaller collection is the same.

## Disadvantages of Hash Tables

Hash tables are more difficult to implement than other collection types. You have to choose a hashing function that will not negate your reasons for using the hash table over other collection types. Some hashing functions could cause a lot of collisions, which would affect the performance of your hash table. Choosing a good hashing function can depend on you knowing ahead of time what type of data you will be adding. Also, if your hashing function is really complex, the application could spend a lot of valuable time executing it.

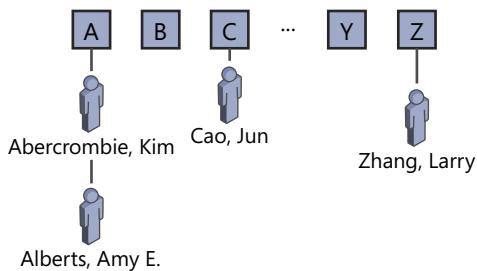
Depending on the implementation of the hashing function, other associative array types might have better performance than a hash table implementation. Make sure you read the documentation of the implementation you plan to use to see if it will perform well in your scenario.

## Understanding Hash Table Buckets

Suppose you have a list of 2,600 phone numbers that are each associated with a name. Each name is in the form "*<Last Name>, <First Name>*" and each phone number is in the form "(###) ###-####". Using a name to find a phone number in the list would take  $O(n)$  operations because you would have to search from the beginning of the list to the end. In real life, if you had to find a phone number associated with a name, you would try to narrow down the search by alphabetizing all of the names according to their last names and searching from there. So imagine that you have 26 buckets, labeled A–Z. Each bucket represents the first letter in a last name.



Now you put each name in the bucket that is labeled with the first letter in the last name.



Your search has now decreased from the entire list to only the name that begins with the first letter of the last name you are looking for. This is the concept behind hash table buckets. Buckets are a way of breaking up a large set of data into a smaller set. The process of mapping a key—in this case, a name—to an index is done by what is called a *hashing function*. An example hashing function from the preceding example would be like the following.

**C#**

```
static int NameToIndex(string name)
{
    // Trim the name and convert it to lower case
    name = name.Trim().ToLower();

    // Subtract 'a' from the last initial to create an index from
    // 0 to 25
    return (int)name[0] - (int)'a';
}
```

**Visual Basic**

```
Function NameToIndex(ByVal name As String) As Integer
    ' Trim the name and convert it to lower case
    name = name.Trim().ToLower()

    ' Subtract 'a' from the last initial to create an index from
    ' 0 to 25
    Return Asc(name(0)) - Asc("a"c)
End Function
```



**Note** The *NameToIndex* method is provided to demonstrate the concept of hash functions. You should research other hashing functions if you want to hash names.

The following example demonstrates how to use the function.

**C#**

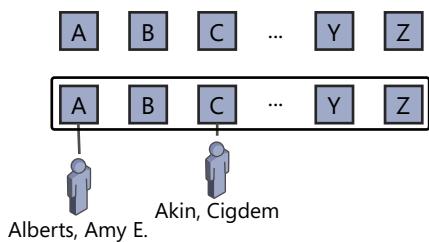
```
string[] buckets = new string[26];
buckets[NameToIndex("Akin, Cigdem")] = "(901) 555-0166";
buckets[NameToIndex("Zimprich, Karin")] = "(901) 555-0177";
```

**Visual Basic**

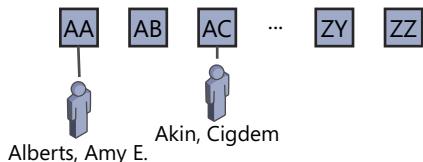
```
Dim buckets As String() = New String(25) {}
buckets(NameToIndex("Akin, Cigdem")) = "(901) 555-0166"
buckets(NameToIndex("Zimprich, Karin")) = "(901) 555-0177"
```

## Understanding Hash Table Collisions

You may have realized a problem with the number of buckets. Out of 2,600 names, you can only have 26 buckets to put them in. If you tried to associate “*Zhang, Larry*” with “(901) 555-0133”, you would find his name at the same slot as “*Zimprich, Karin*” because they both start with the letter Z. This is what is referred to as a collision. A *collision* is when two different keys have the same hash value. To solve this, you can pick a different hashing function that resolves this issue, such as using the first name as well as the last name. With this approach, you would have an additional 26 buckets for the first name after you obtain the last name bucket. Each last name bucket would contain 26 first name buckets.



To index this, we would have to flatten the buckets.



So bucket AA would be used for a person whose last name and first name begin with the letter A. Bucket AB would be used for a person whose last name begins with A and first name begins with B. To convert the name to an index, you can use the following method.

### C#

```
static int NameToIndex(string name)
{
    // Trim the name and convert it to lower case
    name = name.Trim().ToLower();

    // Subtract 'a' from the last initial to create an index from
    // 0 to 25
    int firstbucket = (int)name[0] - (int)'a';

    int lastbucket = 0;
    int indexOfLastName = name.IndexOf(", ");

    // Make the last bucket 0 if there isn't a first name
    if (indexOfLastName >= 0)
    {
        // Subtract 'a' from the first initial to create an index from
        // 0 to 25
        lastbucket = (int)name[indexOfLastName + 2] - (int)'a';
    }

    return lastbucket + firstbucket * 26;
}
```

### Visual Basic

```
Function NameToIndex(ByVal name As String) As Integer
    ' Trim the name and convert it to lower case
    name = name.Trim().ToLower()
```

```
' Subtract 'a' from the last initial to create an index from
' 0 to 25
Dim firstbucket As Integer = Asc(name(0)) - Asc("a"c)

Dim lastbucket As Integer = 0
Dim indexOfLastName As Integer = name.IndexOf(", ")

' Make the last bucket 0 if there isn't a first name
If (indexOfLastName >= 0) Then
    ' Subtract 'a' from the first initial to create an index from
    ' 0 to 25
    lastbucket = Asc(name(indexOfLastName + 2)) - Asc("a"c)
End If

Return lastbucket + firstbucket * 26

End Function
```



**Note** The `NameToIndex` method is provided to demonstrate the concept of hash functions. You should research other hashing functions if you want to hash names.

The hashing function now uses the first initial of the first and last name to compute the index. To use this method, you would do as follows.

### C#

```
string[] buckets = new string[26 * 26];
buckets[NameToIndex("Akin, Cigdem")] = "(901) 555-0166";
buckets[NameToIndex("Zimprich, Karin")] = "(901) 555-0177";
buckets[NameToIndex("Zhang, Larry")] = "(901) 555-0133" ;
```

### Visual Basic

```
Dim buckets As String() = New String(26 * 26 - 1) {}
buckets(NameToIndex("Akin, Cigdem")) = "(901) 555-0166"
buckets(NameToIndex("Zimprich, Karin")) = "(901) 555-0177"
buckets(NameToIndex("Zhang, Larry")) = "(901) 555-0133"
```

You may have realized that you still have an issue. If you associate the name “*Zakardissnehf, Kerstin*” with the phone number “(901) 555-0122”, you would overwrite “*Zimprich, Karin*”. You could pick the next initial and then the next and so on, but you would eventually run out of memory or negate your reason for picking a hash table. So at some point, you have collision unless you can implement a hashing function that puts only one name in a bucket. Because first and last names are not unique in the world, you may find it impossible to find a hashing function that doesn’t contain collisions for a dynamic list. However, there are multiple ways to solve the problem of collision. The next section explains how to solve it using chaining.

## Understanding Chaining

At some point, two different keys will have the same hash value. To solve this, you can use chaining. *Chaining* solves the hash collision problem by implementing a linked list at each bucket. You can then search using the method described earlier for association lists but at the bucket level instead of the complete list. To implement this, you need to create a data type to store your key value pair, as in the following.

### C#

```
struct KVPair
{
    public string Name;
    public string PhoneNumber;
    public KVPair(string name, string phoneNumber)
    {
        Name = name;
        PhoneNumber = phoneNumber;
    }
}
```

### Visual Basic

```
Structure KVPair
    Public Name As String
    Public PhoneNumber As String
    Public Sub New(ByVal name As String, ByVal phoneNumber As String)
        Me.Name = name
        Me.PhoneNumber = phoneNumber
    End Sub
End Structure
```

Each bucket will be a *SingleLinkedList(KVPair)* instead of a *string*. So you need a method such as the one that follows for adding values to the bucket.

### C#

```
static void Associate(SingleLinkedList<KVPair>[] buckets, string name, string phoneNumber)
{
    int index = NameToIndex(name);

    if (buckets[index] == null)
    {
        buckets[index] = new SingleLinkedList<KVPair>();
        buckets[index].AddToBeginning(new KVPair(name, phoneNumber));
        return;
    }

    SingleLinkedList<KVPair> bucket = buckets[index];
    for (SingleLinkedListNode<KVPair> curr = bucket.Head; curr != null; curr = curr.Next)
    {
        if (StringComparer.CurrentCulture.Compare(curr.Data.Name, name) == 0)
```

```

    {
        throw new InvalidOperationException
            ("The specified key has already been associated with a value");
    }
}
bucket.AddToBeginning(new KVPair(name, phoneNumber));
}

```

### Visual Basic

```

Sub Associate(ByVal buckets As SingleLinkedList(Of KVPair)(), _
              ByVal name As String, _
              ByVal phoneNumber As String)
    Dim index As Integer = NameToIndex(name)

    If (buckets(index) Is Nothing) Then
        buckets(index) = New SingleLinkedList(Of KVPair)()
        buckets(index).AddToBeginning(New KVPair(name, phoneNumber))
        Return
    End If

    Dim bucket As SingleLinkedList(Of KVPair) = buckets(index)
    Dim curr As SingleLinkedListNode(Of KVPair) = bucket.Head
    While (curr IsNot Nothing)
        If (StringComparer.CurrentCulture.Compare(curr.Data.Name, name) = 0) Then
            Throw New InvalidOperationException _
                ("The specified key has already been associated with a value")
        End If
        curr = curr.Next
    End While
    bucket.AddToBeginning(New KVPair(name, phoneNumber))
End Sub

```

The *Associate* method creates a *SingleLinkedList(KVPair)* for a bucket if one doesn't already exist. The *Associate* method then checks to see if the key has already been added to the list. If it hasn't, it will add the key value pair to the list, otherwise it will throw an exception. The following code demonstrates this approach.

### C#

```

SingleLinkedList<KVPair> []buckets = new SingleLinkedList<KVPair>[26 * 26];

Associate(buckets, "Akin, Cigdem", "(901) 555-0166");
Associate(buckets, "Zimprich, Karin", "(901) 555-0177");
Associate(buckets, "Zhang, Larry", "(901) 555-0133");
Associate(buckets, "Zakardissnehf, Kerstin", "(901) 555-0122");

```

### Visual Basic

```

Dim buckets() As New SingleLinkedList(Of KVPair)(26 * 26 - 1) {}

Associate(buckets, "Akin, Cigdem", "(901) 555-0166")
Associate(buckets, "Zimprich, Karin", "(901) 555-0177")
Associate(buckets, "Zhang, Larry", "(901) 555-0133")
Associate(buckets, "Zakardissnehf, Kerstin", "(901) 555-0122")

```

## Understanding *Object.GetHashCode*

The Microsoft .NET Framework already implements a *GetHashCode* method in every object that returns a hash code. This method can be used to create a hash code, but you should keep in mind the following:

- The hash code from *GetHashCode* isn't guaranteed to be unique. That means that two objects that are not equal could share the same hash code. However, two objects that are equal should share the same hash code.
- An object could have a different hash code with different versions of the .NET Framework if the application were restarted. In other words, if you persist an object with its hash code, there is no guarantee that it will have the same hash code when the application restarts and loads the object from persistent storage.
- Implementers of an object that is going to be used as a key should override the default behavior of the *GetHashCode* method. If they do not, or if their implementation isn't sufficient enough for your hash table implementation, you should create a custom one using the *System.Collections.IEqualityComparer* interface.
- When implementing *GetHashCode*, you should base the hash code off of at least one of the fields of the instance. A checksum of all of the fields used for comparing it against other objects of the same type is a good choice. That way, two instances that are equal would return the same hash code.

## Picking a Hashing Function

You need to try to uniformly distribute the keys over  $n$  buckets. The formula that will be used in *AssociativeArrayHT(TKey, TValue)* is the following.

$$h(k) = c(k) \bmod n$$

This is known as the *division method*. The division method is not the only method available and is not considered the best for all cases, however, it is easy to follow and you can find a lot of documentation about it.

In the division method,  $c$  is a method that converts the key to a numeric value. You'll use *Object.GetHashCode* for method  $c$ . You then use the modulus operation to uniformly distribute the keys over  $n$  buckets. The modulus operator always returns a number from  $0-(n-1)$ . This is because the modulus operator returns the remainder of the division of one number by another. However, if  $c(k)$  returns a negative number, the modulus operator returns a negative number. The value returned from *Object.GetHashCode* will be AND with the *0xFFFFFFFF*. (The AND operation is discussed in the *BitArray* sections in Chapter 5, "Generic and Support Collections.") This removes the sign bit from the value returned from the *Object.GetHashCode* method, which guarantees the number is a positive number.

## Picking a Good Number of Buckets

You can compute a hash code in many ways, but you will be focusing on one of the simplest and easiest to understand. To compute a hash code, some methods take the fields that change and multiply them by a *constant*. This multiplier needs to be coprime with the number of buckets you pick. *Coprime* means that the two numbers have no common positive factors other than 1. So,  $m$  (*multiplier*) should be coprime to  $n$  (*number of buckets*). Here is a simple way of calculating a string hash code that can be found on the Internet.

### C#

```
class MyStringComparer : IEqualityComparer<string>
{
    const int CONSTANT = 31;

    public bool Equals(string x, string y)
    {
        return StringComparer.CurrentCulture.Equals(x,y);
    }

    public int GetHashCode(string obj)
    {
        if (string.IsNullOrEmpty(obj))
        {
            return 0;
        }

        int hashCode = 0;
        for (int i = 0; i < obj.Length; ++i)
        {
            hashCode = CONSTANT * hashCode + (int)obj[i];
        }

        return hashCode;
    }
}
```

### Visual Basic

```
Class MyStringComparer
    Implements IEqualityComparer(Of String)

    Const CONSTANT As Integer = 31

    Public Function Equals(ByVal x As String, ByVal y As String) As Boolean _
        Implements IEqualityComparer(Of String).Equals
        Return StringComparer.CurrentCulture.Equals(x, y)
    End Function

    Public Function GetHashCode(ByVal obj As String) As Integer _
        Implements IEqualityComparer(Of String).GetHashCode
        If (String.IsNullOrEmpty(obj)) Then
            Return 0
        End If
```

```

        Dim hashCode As Integer = 0
        For i As Integer = 0 To obj.Length - 1
            hashCode = CONSTANT * hashCode + Asc(obj(i))
        Next

        Return hashCode
    End Function
End Class

```



**Note** This is also how you would override the default *GetHashCode* method used in the *Dictionary(TKey,TValue)* and *AssociativeArrayHT(TKey,TValue)* classes.

In the preceding example, *m* is *CONSTANT* or 31. So the *n* you pick needs to coprime with *m*. To understand why, look at the following simplified example that doesn't follow the rule. The code multiplies *m* by *i* to *samples*, takes the modulus of *n*, and then increments the number of items in that bucket.

#### C#

```

int m = 31;
int hashCode = 0;
int n = 62;
int samples = 1000;
int[] buckets = new int[n];
for (int i = 0; i < samples; ++i)
{
    hashCode = m * i;
    ++buckets[hashCode % n];
}
for (int i = 0; i < n; ++i)
{
    if (buckets[i] != 0)
    {
        Console.WriteLine("buckets[{0}] = {1}%", i, buckets[i] * 100 / samples);
    }
}

```

#### Visual Basic

```

Dim m As Integer = 31
Dim hashCode As Integer = 0
Dim n As Integer = 62
Dim samples As Integer = 1000
Dim buckets As Integer() = New Integer(n - 1) {}
For i As Integer = 0 To samples - 1
    hashCode = m * i
    buckets(hashCode Mod n) += 1
Next

For i As Integer = 0 To n - 1
    If (buckets(i) >> 0) Then
        Console.WriteLine("buckets[{0}] = {1}%", i, buckets(i) * 100 / samples)
    End If
Next

```

## Output

```
buckets[0] = 50%
buckets[31] = 50%
```

You can see how out of 1,000 simulated samples, buckets 0 and 31 received all of the items. This is because both numbers are divisible by 31, which makes them not coprime. So how can you guarantee that both numbers are coprime to each other? You can have  $m$  and  $n$  both be prime numbers. *Prime numbers* are numbers that are only divisible by 1 and themselves. Even numbers except 2 are never prime because they can be divisible by 2. Numbers that end with 0 and 5 except 5 are never prime because they are divisible by 5. So two prime numbers cannot have any common factors. This makes them perfect for  $n$  and  $m$ . The only problem is that you can control  $n$  but not necessarily  $m$  because you normally will not write the *GetHashCode* method for it. You can still pick a prime number for  $n$  that will work, assuming the writer of *GetHashCode* used a prime number or a method that doesn't use a multiplier. The same example follows, but it has a prime number for  $n$  and  $m$ .

## C#

```
int m = 31;
int hashCode = 0;
int n = 11;
int samples = 1000;
int[] buckets = new int[n];
for (int i = 0; i < samples; ++i)
{
    hashCode = m * i;
    ++buckets[hashCode % n];
}
for (int i = 0; i < n; ++i)
{
    if (buckets[i] != 0)
    {
        Console.WriteLine("buckets[{0}] = {1}%", i, buckets[i] * 100 / samples);
    }
}
```

## Visual Basic

```
Dim m As Integer = 31
Dim hashCode As Integer = 0
Dim n As Integer = 11
Dim samples As Integer = 1000
Dim buckets As Integer() = New Integer(n - 1) {}
For i As Integer = 0 To samples - 1
    hashCode = m * i
    buckets(hashCode Mod n) += 1
Next

For i As Integer = 0 To n - 1
    If (buckets(i) <> 0) Then
        Console.WriteLine("buckets[{0}] = {1}%", i, buckets(i) * 100 / samples)
    End If
Next
```

### Output

```
buckets[0] = 9%
buckets[1] = 9%
buckets[2] = 9%
buckets[3] = 9%
buckets[4] = 9%
buckets[5] = 9%
buckets[6] = 9%
buckets[7] = 9%
buckets[8] = 9%
buckets[9] = 9%
buckets[10] = 9%
```

Now the items are evenly distributed across  $n$  buckets because  $n$  and  $m$  are now coprimes. So as long as *MyStringComparer.GetHashCode* returns a uniform distribution of hash codes, the items will be uniformly distributed across our buckets.

One problem with using prime numbers for  $n$  is that they are very expensive to compute at run time. You could create a lookup table to contain the prime numbers instead of computing them at run time. You can also find a good predefined list of prime numbers by searching the Internet and then running stats on your implementation to see which one performs best for your scenario.

If  $m$  is a prime number, you can also use the even rule we talked about earlier in this section. A prime number can never have an even number as a factor, but an even number can have a prime number as a factor; for example, the even number 14 has the prime numbers 7 and 2 as factors. All  $2^i$ , when  $i$  is greater than 2, has the prime number 2 as a factor and another number that isn't prime. An even number times an odd number is always even, and an odd number times an odd number is always odd. Using this information, you could let  $n$  be a power of 2, assuming that the writer of the *GetHashCode* method didn't use 2 for  $m$  or an even number. Because all even numbers are divisible by 2,  $n$  and  $m$  would at least have a common factor of 2 if  $m$  was even and  $n$  was a power of 2. So,  $2^i$  will never be a prime number or an odd number when  $i$  is greater than 1 and can be used for  $n$  when  $m$  isn't even.

## Implementing Your Internal Storage

The easiest way to implement a bucket is to use our existing linked list class to represent each bucket. When you add an item to a bucket, you add it to the beginning of the list. You search the list for an existing entry when you need to associate an existing key to a new value. When it is time to increase the size of the hash table, you can hold the old buckets in a temporary variable, traverse each entry in the old buckets, and then assign the entry to the new buckets based on their new hashed values. Sounds good until you think about the disadvantage of a linked list over an array. Linked lists use more memory than arrays. So if you have 200 buckets, you would have 200 instances of the linked list class.

The other issue with using the linked list classes is that they are not written with high performance in mind, which is why you went through the process of implementing a linked list and other collection classes as well. Implementing them helps you to understand where the bottlenecks of an implementation are and what to do to improve them.

If you are presented with  $n$  items to add to your hash table, the best case scenario would be  $n$  buckets with one item in each bucket. The worst case scenario would be one bucket with  $n$  items in it. More than likely, you will be somewhere between the two scenarios unless you know ahead of time all of the items you will be adding—or the guy who is quitting next week creates your hashing function.

The number of buckets will also be used in the hash function. So, every time you increase or decrease the number of buckets, you need to traverse all items and calculate their new hash positions. Traversing all items and calculating their hash positions is an expensive operation. Also, removing the item from one list involves fixing the node links, so the potential problems identified are as follows:

- You need a linked list for each bucket.
- You need a bucket for each possible item.
- You need to avoid having to rehash the table.
- You need to watch memory usage.
- You need to allow quick insertions and deletions.

An array sounds like a good implementation until you have to rehash the table or do quick insertions and deletions. It has lower memory usage than a linked list, but you still need to create  $n$  arrays. One way to solve this is to use an array to hold the linked list nodes. You would still have problems with creating and increasing the size of the arrays each time an item is added, and shifting items for an insertion and deletion. To combat this, you can create an array for each bucket that holds the maximum number of items for the worst case scenario of  $n$  items. However, if each bucket contains  $n$  items and you have  $n$  buckets, you need to create  $n \times n$  nodes. As stated earlier, the worst case is  $n$  items in one bucket and the best is 1 item in  $n$  buckets. What if there is an easy way to share nodes across buckets? You begin by creating an array of shared nodes, as in the following.

### C#

```
Entry< TKey, TValue>[] m_entries;

// Nested Types
private struct Entry
{
    public TKey Key;
    public TValue Value;
    public int HashCode;
    public int Next;
}
```

**Visual Basic**

```
Private Structure Entry
    Public Key As TKey
    Public Value As TValue
    Public HashCode As Integer
    Public [Next] As Integer
End Structure
```

The *Entry* struct represents a node. The *Next* field references the next node. Each reference is stored as an index to *m\_entries*. This gives you a maximum of *Int32.MaxValue* nodes. If you needed more, you could use an *UInt32*, or if you needed fewer, you could use an *Int16*, an *Uln16*, or even a *Byte*. When you need a new node, it takes a while to traverse *m\_entries* and find a node that isn't being used. You would also have to store a flag that states that the node is free. To solve this, the following fields are added to *AssociativeArrayHT(T)*.

**C#**

```
int m_nextUnusedEntry;
int m_unusedCount;
```

**Visual Basic**

```
Private m_nextUnusedEntry As Integer
Private m_unusedCount As Integer
```

The *m\_nextUnusedEntry* points to the first unused node. That node then points to the next unused one and so on. When it is time to retrieve an unused node, you only have to obtain the node pointed to by *m\_nextUnusedEntry* and then assign *m\_nextUnusedEntry* to the next unused one.

To share the node across buckets, you can create the following fields.

**C#**

```
int[] m_buckets;
int m_capacity;
int m_count;
public const int NULL_REFERENCE = -1;
```

**Visual Basic**

```
Private m_buckets As Integer()
Private m_count As Integer
Private m_capacity As Integer
Public Const NULL_REFERENCE As Integer = -1
```

Each element in *m\_buckets* references the first node in that bucket. The constant *NULL\_REFERENCE* is used to denote an empty list. If a node's *Next* field is equal to *NULL\_REFERENCE*, that node will be the tail of the linked list. The *m\_capacity* field states how many nodes and buckets are present. The *m\_count* field states how many nodes are currently used.

## Getting Started

You can create a class called *AssociativeArrayHT(T)* by using the following.

C#

```
[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(AssociativeArrayDebugView))]
public class AssociativeArrayHT<TKey, TValue>
{
    // Fields
    private int[] m_buckets;
    private int m_capacity;
    private IEqualityComparer<TKey> m_comparer;
    private int m_count;
    private Entry<TKey, TValue>[] m_entries;
    private int m_nextUnusedEntry;
    private int m_unusedCount;
    private int m_updateCode;
    public const int NULL_REFERENCE = -1;
    private readonly int[] PRIME_NUMBERS;

    // Constructors
    public AssociativeArrayHT();
    public AssociativeArrayHT(IEqualityComparer<TKey> comparer);

    // Methods
    public void Add(TKey key, TValue value);
    private void Add(TKey key, TValue value, bool overwrite);
    private int CalculateCapacity(int size);
    public void Clear();
    public bool ContainsKey(TKey key);
    public bool ContainsValue(TValue value);
    private EntryData<TKey, TValue> FindKey(TKey key);
    private EntryData<TKey, TValue> FindValue(TValue value);
    private int CalculateHashCode(TKey key);
    internal bool MoveNext(ref int bucketIndex, ref int entryIndex);
    private void Rehash();
    private bool Remove(EntryData<TKey, TValue> entry);
    public bool Remove(TKey key);
    public bool RemoveValue(TValue value);
    public bool RemoveValue(TValue value, bool allOccurrences);
    public bool TryGetValue(TKey key, out TValue value);

    // Properties
    public int Count { get; }
    public bool IsEmpty { get; }
    public TValue this[TKey key] { get; set; }
    public TKey[] Keys { get; }
    public TValue[] Values { get; }
```

```

// Nested Types
private struct Entry
{
    public TKey Key;
    public TValue Value;
    public int HashCode;
    public int Next;
}

private struct EntryData
{
    private AssociativeArrayHT<TKey, TValue> m_hashtable;
    private int m_index;
    private int m_bucketIndex;
    private int m_previous;
    public static readonly AssociativeArrayHT<TKey, TValue>.EntryData EMPTY;
    public int Index { get; }
    public int BucketIndex { get; }
    public bool IsEmpty { get; }
    public int Previous { get; }
    public int Next { get; }
    public TKey Key { get; }
    public TValue Value { get; set; }
    public EntryData(AssociativeArrayHT<TKey, TValue> hashtable, int index,
                    int previous, int bucketIndex);
    static EntryData();
}
}

```

### Visual Basic

```

<DefaultMember("Item")> _
<DebuggerDisplay("Count={Count}")> _
<DebuggerTypeProxy(GetType(AssociativeArrayDebugView))> _
Public Class AssociativeArrayHT(Of TKey, TValue)

    ' Fields
    Private m_buckets As Integer()
    Private m_capacity As Integer
    Private m_comparer As IEqualityComparer(Of TKey)
    Private m_count As Integer
    Private m_entries As Entry(Of TKey, TValue)()
    Private m_nextUnusedEntry As Integer
    Private m_unusedCount As Integer
    Private m_updateCode As Integer
    Public Const NULL_REFERENCE As Integer = -1
    Private ReadOnly PRIME_NUMBERS As Integer()

    ' Constructors
    Public Sub New()
        Public Sub New(ByVal comparer As IEqualityComparer(Of TKey))

    ' Methods
    Public Sub Add(ByVal key As TKey, ByVal value As TValue)
    Public Sub Add(ByVal key As TKey, ByVal value As TValue, ByVal overwrite As Boolean)
    Private Function CalculateCapacity(ByVal size As Integer) As Integer

```

```
Private Function CalculateHashCode(ByVal key As TKey) As Integer
Public Sub Clear()
Public Function ContainsKey(ByVal key As TKey) As Boolean
Public Function ContainsValue(ByVal value As TValue) As Boolean
Private Function FindKey(ByVal key As TKey) As EntryData(Of TKey, TValue)
Private Function FindValue(ByVal value As TValue) As EntryData(Of TKey, TValue)
Private Function HashFunction(ByVal hashcode As Integer) As Integer
Friend Function MoveNext(ByRef bucketIndex As Integer, ByRef entryIndex As Integer) _
    As Boolean
Private Sub Rehash()
Private Function Remove(ByVal entry As EntryData(Of TKey, TValue)) As Boolean
Public Function Remove(ByVal key As TKey) As Boolean
Public Function RemoveValue(ByVal value As TValue) As Boolean
Public Function RemoveValue(ByVal value As TValue, ByVal allOccurrences As Boolean) _
    As Boolean
Public Function TryGetValue(ByVal key As TKey, ByRef value As TValue) As Boolean

' Properties
Public ReadOnly Property Count As Integer
Public ReadOnly Property IsEmpty As Boolean
Public Default Property Item(ByVal key As TKey) As TValue
Public ReadOnly Property Keys As TKey()
Public ReadOnly Property Values As TValue()

' Nested Types
<StructLayout(LayoutKind.Sequential)> _
Private Structure Entry
    Public Key As TKey
    Public Value As TValue
    Public HashCode As Integer
    Public [Next] As Integer
End Structure

<StructLayout(LayoutKind.Sequential)> _
Private Structure EntryData
    private m_hashtable As AssociativeArrayHT(Of TKey, TValue)
    private m_index As Integer
    private m_bucketIndex As Integer
    private m_previous As Integer
    Public Shared ReadOnly EMPTY As EntryData(Of TKey, TValue)
    Shared Sub New()
        Public ReadOnly Property Index As Integer
        Public ReadOnly Property BucketIndex As Integer
        Public ReadOnly Property IsEmpty As Boolean
        Public ReadOnly Property Previous As Integer
        Public ReadOnly Property [Next] As Integer
        Public ReadOnly Property Key As TKey
        Public Property Value As TValue
        Public Sub New(ByVal hashtable As AssociativeArrayHT(Of TKey, TValue), _
            ByVal index As Integer, _
            ByVal previous As Integer, _
            ByVal bucketIndex As Integer)
    End Sub
End Structure
End Class
```

The *m\_updateCode* field is incremented each time the user modifies the list. The *m\_updateCode* field is used in Chapter 6 to determine whether the collection has changed while the user is iterating over it. It is easier to add it to the code now than change the code in Chapter 6.

The comparer that is used for comparing keys is stored in *m\_comparer*. The *m\_comparer* field is used to check whether a key matches a key stored in the hash table. *PRIME\_NUMBERS* contains a predefined list of prime numbers that can be used so that they do not have to be calculated during run time.

The *EntryData* struct is used to store detailed information about an entry, such as the bucket it belongs to and the previous bucket. Performance-wise, this helps when the hash table needs to remove an entry it has found.

## Creating the Constructor

The *AssociativeArrayHT(T)* class will contain two constructors. One constructor is for creating an empty class, and the other is for specifying a comparer to be used for the key comparisons.

The default constructor for the *AssociativeArrayHT(T)* class allows users to create an empty associative array.

### C#

```
/// <summary>
/// Initializes a new instance of the AssociativeArrayHT(TKey, TValue) class that is empty.
/// </summary>
public AssociativeArrayHT()
{
    m_comparer = EqualityComparer<TKey>.Default;
}
```

### Visual Basic

```
''' <summary>
''' Initializes a new instance of the AssociativeArrayHT(TKey, TValue) class that is empty.
''' </summary>
Public Sub New()
    m_comparer = EqualityComparer(Of TKey).Default
End Sub
```

The next constructor allows the user to specify the comparer that will be used to do key comparisons.

### C#

```
/// <summary>
/// Initializes a new instance of the AssociativeArrayHT(TKey, TValue) class
/// that is empty and uses the specified comparer.
/// </summary>
/// <param name="comparer">The comparer to use for the keys.</param>
```

```
public AssociativeArrayHT(IEqualityComparer<TKey> comparer)
{
    if (comparer == null)
    {
        throw new ArgumentNullException("comparer");
    }

    m_comparer = comparer;
}
```

### Visual Basic

```
''' <summary>
''' Initializes a new instance of the AssociativeArrayHT(TKey,TValue) class
''' that is empty and uses the specified comparer.
''' </summary>
''' <param name="comparer">The comparer to use for the keys.</param>
Public Sub New(ByVal comparer As IEqualityComparer(Of TKey))
    If (comparer Is Nothing) Then
        Throw New ArgumentNullException("comparer")
    End If

    m_comparer = comparer
End Sub
```

## Allowing Users to Associate Values with Keys

Users need the ability to associate values with keys, and to reassign keys as well. They will be able to accomplish this using the *Add* method and *Item* property. The *Add* method and *Item* property are defined as follows.

### C#

```
/// <summary>
/// Adds the key value pair to the AssociativeArrayAL(TKey,TValue).
/// </summary>
/// <param name="key">The key to associate with the value.</param>
/// <param name="value">The value to add.</param>
public void Add(TKey key, TValue value)
{
    Add(key, value, false);
}

/// <summary>
/// Sets the value at the specified key.
/// </summary>
/// <param name="key">The key to use for finding the value.</param>
/// <returns>The value associated with the specified key.</returns>
public TValue this[TKey key]
{
    set
    {
        Add(key, value, true);
    }
}
```

### Visual Basic

```

''' <summary>
''' Adds the key value pair to the AssociativeArrayHT(TKey,TValue).
''' </summary>
''' <param name="key">The key to associate with the value.</param>
''' <param name="value">The value to add.</param>
Public Sub Add(ByVal key As TKey, ByVal value As TValue)
    Add(key, value, False)
End Sub
''' <summary>
''' Gets or sets the value at the specified key.
''' </summary>
''' <param name="key">The key to use for finding the value.</param>
''' <returns>The value associated with the specified key.</returns>
Default Public Property Item(ByVal key As TKey) As TValue
    Set(ByVal value As TValue)
        Add(key, Value, True)
    End Set
End Property

```

Both call the *Add(TKey,TValue,bool)* method, which is defined as follows.

### C#

```

/// <summary>
/// Adds the key value pair to the AssociativeArrayHT(TKey,TValue)
/// </summary>
/// <param name="key">The key to associate with the specified value.</param>
/// <param name="value">The value to add to the AssociativeArrayHT(TKey,TValue).</param>
/// <param name="overwrite">
/// True if the value should be overwritten if it exist, false if an error should be thrown.
/// </param>
void Add(TKey key, TValue value, bool overwrite)
{
    EntryData entry = FindKey(key);
    if (!entry.IsEmpty)
    {
        if (!overwrite)
        {
            throw new InvalidOperationException
                ("The specified key is already present in the array");
        }
        else
        {
            entry.Value = value;
        }
    }
    return;
}

if (m_unusedCount <= 0)
{
    Rehash();
}

```

```

int uHashCode = CalculateHashCode(key);

int bucketIndex = HashFunction(uHashCode);

int thisEntry = m_nextUnusedEntry;

// Pop the first unused entry off of the unused bucket
m_nextUnusedEntry = m_entries[m_nextUnusedEntry].Next;
--m_unusedCount;

++m_count;

m_entries[thisEntry].Key = key;
m_entries[thisEntry].Value = value;
m_entries[thisEntry].HashCode = uHashCode;
m_entries[thisEntry].Next = m_buckets[bucketIndex];

m_buckets[bucketIndex] = thisEntry;

++m_updateCode;
}

```

### Visual Basic

```

''' <summary>
''' Adds the key value pair to the AssociativeArrayHT(TKey, TValue)
''' </summary>
''' <param name="key">The key to associate with the specified value.</param>
''' <param name="value">The value to add to the AssociativeArrayHT(TKey, TValue).</param>
''' <param name="overwrite">
''' True if the value should be overwritten if it exist, false if an error should be thrown.
''' </param>
Sub Add(ByVal key As TKey, ByVal value As TValue, ByVal overwrite As Boolean)
    Dim entry As EntryData = FindKey(key)
    If (Not entry.IsEmpty) Then
        If (Not overwrite) Then
            Throw New InvalidOperationException _
                ("The specified key is already present in the array")
        Else
            entry.Value = value
        End If
        Return
    End If
    If (m_unusedCount <= 0) Then
        Rehash()
    End If
    Dim uHashCode As Integer = CalculateHashCode(key)
    Dim bucketIndex As Integer = HashFunction(uHashCode)
    Dim thisEntry As Integer = m_nextUnusedEntry

```

```

' Pop the first unused entry off of the unused bucket
m_unusedEntry = m_entries(m_unusedEntry).Next
m_unusedCount -= 1

m_count += 1

m_entries(thisEntry).Key = key
m_entries(thisEntry).Value = value
m_entries(thisEntry).HashCode = uHashCode
m_entries(thisEntry).Next = m_buckets(bucketIndex)

m_buckets(bucketIndex) = thisEntry

m_updateCode += 1
End Sub

```

The *Add(TKey, TValue, bool)* method uses the *FindKey* method to see whether the key has already been added. If the key is already present and the user wasn't trying to reassign the key, an *InvalidOperationException* will be thrown. If the key exists and the user was trying to reassign the key, the method reassigns the key and then returns. If there isn't enough room to add the key value pair, the method rehashes the table as described in the following code. The method then adds the key/value pair to the beginning of the list and increments the update code.

### C#

```

void Rehash()
{
    m_capacity = CalculateCapacity(m_capacity + 1);

    Entry[] oldData = m_entries;
    int[] oldBuckets = m_buckets;

    m_entries = new Entry[m_capacity];

    // Create new buckets and set them to null
    m_buckets = new int[m_capacity];
    for (int bucketIndex = 0; bucketIndex < m_buckets.Length; ++bucketIndex)
    {
        m_buckets[bucketIndex] = NULL_REFERENCE;
    }

    int unusedStart = 0;

    if (oldData != null)
    {
        // Copy the old bucket to the new one
        Array.Copy(oldData, m_entries, oldData.Length);

        // All entries that were just created are unused.
        // They all begin at index oldData.Length
        unusedStart = oldData.Length;
    }
}

```

```
if (_unusedCount <= 0)
{
    // Assign the next deleted entry to the beginning of the free data
    _nextUnusedEntry = oldData.Length;
}
else
{
    // Find the last deleted entry and set its next to the beginning
    // of the appended unused entries
    for (int i = _nextUnusedEntry; i != NULL_REFERENCE; i = _entries[i].Next)
    {
        if (_entries[i].Next == NULL_REFERENCE)
        {
            _entries[i].Next = oldData.Length;
            break;
        }
    }
}

// Add all of the newly created entry to the unused count
_UNUSEDCount += (_capacity - oldData.Length);
}
else
{
    _unusedCount = _capacity;
    _nextUnusedEntry = 0;
}

// Set the next pointer of all unused entries to the next unused entry
for (int i = unusedStart; i < _capacity - 1; ++i)
{
    _entries[i].Next = i + 1;
}
_entries[_capacity - 1].Next = NULL_REFERENCE;

// Recalculate everyone's location in the hash table
if (oldBuckets != null)
{
    // Traverses the old buckets and move the entries to their new buckets
    // in the new hash table.
    for (int bucketIndex = 0; bucketIndex < oldBuckets.Length; ++bucketIndex)
    {
        int index = oldBuckets[bucketIndex];

        // Traverse each entry in this bucket
        while (index != NULL_REFERENCE)
        {
            int nextIndex = _entries[index].Next;

            // Make sure the hash code is positive.
            int newBucketIndex = HashFunction(_entries[index].HashCode);

            // Set the next of the current entry equal to the first entry in the bucket
            _entries[index].Next = _buckets[newBucketIndex];
        }
    }
}
```

```

        // Set this entry as the first one in the bucket
        m_buckets[newBucketIndex] = index;

        index = nextIndex;
    }
}

++m_updateCode;
}

```

### Visual Basic

```

Private Sub Rehash()
    m_capacity = CalculateCapacity(m_capacity + 1)

    Dim oldData As Entry() = m_entries
    Dim oldBuckets As Integer() = m_buckets

    m_entries = New Entry(m_capacity - 1) {}

    ' Create new buckets and set them to null
    m_buckets = New Integer(m_capacity - 1) {}
    For bucketIndex As Integer = 0 To m_buckets.Length - 1
        m_buckets(bucketIndex) = NULL_REFERENCE
    Next

    Dim unusedStart As Integer = 0

    If (oldData IsNot Nothing) Then
        ' Copy the old bucket to the new one
        Array.Copy(oldData, m_entries, oldData.Length)

        ' All entries that were just created are unused.
        ' They all begin at index oldData.Length
        unusedStart = oldData.Length

        If (m_unusedCount <= 0) Then
            ' Assign the next deleted entry to the beginning of the free data
            m_nextUnusedEntry = oldData.Length
        Else
            ' Find the last deleted entry and set its next to the beginning
            ' of the appended unused entries
            Dim i As Integer = m_nextUnusedEntry
            While (i <> NULL_REFERENCE)

```

```
If (m_entries(i).Next = NULL_REFERENCE) Then
    m_entries(i).Next = oldData.Length
    Exit While
End If
i = m_entries(i).Next
End While
End If

' Add all of the newly created entry to the unused count
m_unusedCount += (m_capacity - oldData.Length)
Else
    m_unusedCount = m_capacity
    m_nextUnusedEntry = 0
End If

' Set the next pointer of all unused entries to the next unused entry
For i As Integer = unusedStart To m_capacity - 2
    m_entries(i).Next = i + 1
Next
m_entries(m_capacity - 1).Next = NULL_REFERENCE

' Recalculate everyone's location in the hash table
If (oldBuckets IsNot Nothing) Then
    ' Traverses the old buckets and move the entries to their new buckets
    ' in the new hash table.
    For bucketIndex As Integer = 0 To oldBuckets.Length - 1
        Dim index As Integer = oldBuckets(bucketIndex)

        ' Traverse each entry in this bucket
        While (index <> NULL_REFERENCE)
            Dim nextIndex As Integer = m_entries(index).Next

            ' Make sure the hash code is positive.
            Dim newBucketIndex As Integer = HashFunction(m_entries(index).HashCode)

            ' Set the next of the current entry equal to the first entry in the bucket
            m_entries(index).Next = m_buckets(newBucketIndex)

            ' Set this entry as the first one in the bucket
            m_buckets(newBucketIndex) = index

            index = nextIndex
        End While
    Next

    End If

    m_updateCode += 1
End Sub
```

The method uses the *CalculateCapacity* method defined in the following code to calculate the next capacity. After the capacity has been calculated, the old buckets and entries are saved for the rehashing operation later. The new buckets and entries are then created, and each old entry is traversed using the old buckets so that it can be added to the new hash table.

### C#

```
/// <summary>
/// Locates the next prime number that should be used for the capacity.
/// </summary>
/// <param name="size">The minimum size needed.</param>
/// <returns>A prime number large enough to hold the specified number of items.</returns>
int CalculateCapacity(int size)
{
    for (int i = 0; i < PRIME_NUMBERS.Length; ++i)
    {
        if (size <= PRIME_NUMBERS[i])
        {
            return PRIME_NUMBERS[i];
        }
    }

    return int.MaxValue;
}
```

### Visual Basic

```
''' <summary>
''' Locates the next prime number that should be used for the capacity.
''' </summary>
''' <param name="size">The minimum size needed.</param>
''' <returns>A prime number large enough to hold the specified number of items.</returns>
Private Function CalculateCapacity(ByVal size As Integer) As Integer
    For i As Integer = 0 To PRIME_NUMBERS.Length - 1
        If (size <= PRIME_NUMBERS(i)) Then
            Return PRIME_NUMBERS(i)
        End If
    Next

    Return Integer.MaxValue
End Function
```

The *CalculateCapacity* method searches *PRIME\_NUMBERS* for the next prime number that can hold the specified size.

## Allowing Users to Remove Keys

Keys also need to be removed when they are no longer needed. The *Clear*, *Remove*, and *RemoveValue* methods are used to do this.

The *Clear* method removes all key/value pairs from the collection and is defined as follows.

**C#**

```
/// <summary>
/// Removes all items from the AssociativeArrayHT(TKey, TValue).
/// </summary>
public void Clear()
{
    // Set each bucket to empty
    for (int i = 0; i < m_buckets.Length; ++i)
    {
        m_buckets[i] = NULL_REFERENCE;
    }

    // Point each entry to the next entry
    for (int i = 0; i < m_capacity - 1; ++i)
    {
        m_entries[i].Key = default(TKey);
        m_entries[i].Value = default(TValue);
        m_entries[i].HashCode = 0;
        m_entries[i].Next = i + 1;
    }
    m_entries[m_capacity - 1].Next = NULL_REFERENCE;

    // Set the first unused entry to the first entry
    m_nextUnusedEntry = 0;

    m_unusedCount = m_capacity;
    m_count = 0;

    ++m_updateCode;
}
```

**Visual Basic**

```
''' <summary>
''' Removes all items from the AssociativeArrayHT(TKey, TValue).
''' </summary>
Public Sub Clear()
    ' Set each bucket to empty
    For i As Integer = 0 To m_buckets.Length - 1
        m_buckets(i) = NULL_REFERENCE
    Next

    ' Point each entry to the next entry
    For i As Integer = 0 To m_capacity - 2
        m_entries(i).Key = CType(Nothing, TKey)
        m_entries(i).Value = CType(Nothing, TValue)
        m_entries(i).HashCode = 0
        m_entries(i).Next = i + 1
    Next
    m_entries(m_capacity - 1).Next = NULL_REFERENCE
```

```

    ' Set the first unused entry to the first entry
    m_unusedEntry = 0

    m_unusedCount = m_capacity
    m_count = 0

    m_updateCode += 1
End Sub

```

The *Clear* method empties all buckets by assigning them to *NULL\_REFERENCE*. It then links all nodes to each other and assigns *m\_nextUnusedEntry* to the first node.

The *Remove* method removes the key/value pair associated with the specified key and is defined as follows.

#### C#

```

/// <summary>
/// Removes the specified key from the AssociativeArrayHT(TKey,TValue).
/// </summary>
/// <param name="key">The key to remove from the AssociativeArrayHT(TKey,TValue).</param>
/// <returns>True if the key was removed, false otherwise.</returns>
public bool Remove(TKey key)
{
    EntryData entry = FindKey(key);

    if (entry.IsEmpty)
    {
        return false;
    }

    return Remove(entry);
}

```

#### Visual Basic

```

''' <summary>
''' Removes the specified key from the AssociativeArrayHT(TKey,TValue).
''' </summary>
''' <param name="key">The key to remove from the AssociativeArrayHT(TKey,TValue).</param>
''' <returns>True if the key was removed, false otherwise.</returns>
Public Function Remove(ByVal key As TKey) As Boolean
    Dim entry As EntryData = FindKey(key)

    If (entry.IsEmpty) Then
        Return False
    End If

    Return Remove(entry)
End Function

```

The *Remove(EntryData)* and *FindKey* methods are discussed later in this chapter.

The *RemoveValue* method removes the key/value pair that contains the specified value and is defined as follows.

C#

```
/// <summary>
/// Removes the first occurrence of the specified value.
/// </summary>
/// <param name="value">The value to remove.</param>
/// <returns>
/// True if the value was removed, false if it wasn't present
/// in the AssociativeArrayHT(TKey,TValue).
/// </returns>
public bool RemoveValue(TValue value)
{
    return RemoveValue(value, false);
}

/// <summary>
/// Removes the specified value.
/// </summary>
/// <param name="value">The value to remove.</param>
/// <param name="allOccurrences">
/// True if all occurrences of the value should be removed, false if not.
/// </param>
/// <returns>
/// True if the value was removed, false if it wasn't present
/// in the AssociativeArrayHT(TKey,TValue).
/// </returns>
public bool RemoveValue(TValue value, bool allOccurrences)
{
    bool removed = false;

    EntryData entry = FindValue(value);
    while (!entry.IsEmpty)
    {
        removed = Remove(entry) || removed;

        if (!allOccurrences)
        {
            return removed;
        }

        entry = FindValue(value);
    }

    return removed;
}
```

### Visual Basic

```

''' <summary>
''' Removes the first occurrence of the specified value.
''' </summary>
''' <param name="value">The value to remove.</param>
''' <returns>
''' True if the value was removed, false if it wasn't present
''' in the AssociativeArrayHT(TKey,TValue).
''' </returns>
Public Function RemoveValue(ByVal value As TValue) As Boolean
    Return RemoveValue(value, False)
End Function

''' <summary>
''' Removes the specified value.
''' </summary>
''' <param name="value">The value to remove.</param>
''' <param name="allOccurrences">
''' True if all occurrences of the value should be removed, false if not.
''' </param>
''' <returns>
''' True if the value was removed, false if it wasn't present
''' in the AssociativeArrayHT(TKey,TValue).
''' </returns>
Public Function RemoveValue(ByVal value As TValue, ByVal allOccurrences As Boolean) _
As Boolean
    Dim removed As Boolean = False

    Dim Entry As EntryData = FindValue(value)
    While (Not Entry.IsEmpty)
        removed = Remove(Entry) Or removed

        If (Not allOccurrences) Then
            Return removed
        End If

        Entry = FindValue(value)
    End While

    Return removed
End Function

```

The *Remove* and *RemoveAll* methods both use the *Remove(EntryData)* method, which is defined as follows.

### C#

```

/// <summary>
/// Removes the entry specified by the user.
/// </summary>
/// <param name="entry">The entry to remove.</param>
/// <returns>True if the entry is removed, false if not.</returns>
bool Remove(EntryData entry)

```

```

{
    if (entry.IsEmpty)
    {
        return false;
    }

    EntryData data = entry;
    if (data.Previous != NULL_REFERENCE)
    {
        // Link the previous and next entry
        m_entries[data.Previous].Next = m_entries[data.Index].Next;
    }
    else
    {
        // The entry is at the beginning of the bucket
        // So set the bucket to reference the next entry
        m_buckets[data.BucketIndex] = m_entries[data.Index].Next;
    }

    // Add the removed entry to the front of the unused bucket.
    m_entries[data.Index].Next = m_nextUnusedEntry;
    m_nextUnusedEntry = data.Index;
    ++m_unusedCount;
    --m_count;

    // Clear the entry
    m_entries[data.Index].Key = default(TKey);
    m_entries[data.Index].Value = default(TValue);
    m_entries[data.Index].HashCode = 0;

    ++m_updateCode;

    return true;
}

```

### Visual Basic

```

''' <summary>
''' Removes the entry associated with the key.
''' </summary>
''' <param name="entry">The entry to remove.</param>
''' <returns>True if the entry is removed, false if not.</returns>
Private Function Remove(ByVal entry As EntryData) As Boolean
    If (entry.IsEmpty) Then
        Return False
    End If

    Dim data As EntryData = DirectCast(entry, EntryData)
    If (data.Previous <> NULL_REFERENCE) Then
        ' Link the previous and next entry
        m_entries(data.Previous).Next = m_entries(data.Index).Next
    Else
        ' The entry is at the beginning of the bucket
        ' So set the bucket to reference the next entry
    End If

```

```

    m_buckets(data.BucketIndex) = m_entries(data.Index).Next
End If

' Add the removed entry to the front of the unused bucket.
m_entries(data.Index).Next = m_nextUnusedEntry
m_nextUnusedEntry = data.Index
m_unusedCount += 1
m_count -= 1

' Clear the entry
m_entries(data.Index).Key = CType(Nothing, TKey)
m_entries(data.Index).Value = CType(Nothing, TValue)
m_entries(data.Index).HashCode = 0

m_updateCode += 1

Return True
End Function

```

The *Remove(EntryData)* method removes the node from the linked list and fixes the surrounding links. The removed node is then added to the linked list at *m\_nextUnusedEntry*.

## Adding Helper Methods and Properties

Users can get the number of items in the collection by using the *Count* property.

### C#

```

/// <summary>
/// Gets the number of items in the AssociativeArrayHT(TKey,TValue).
/// </summary>
public int Count
{
    get { return m_count; }
}

```

### Visual Basic

```

''' <summary>
''' Gets the number of items in the AssociativeArrayHT(TKey,TValue).
''' </summary>
Public ReadOnly Property Count() As Integer
    Get
        Return m_count
    End Get
End Property

```

The *CalculateHashCode* method is used to make sure the hash code retrieved from the comparer is positive.

**C#**

```
int CalculateHashCode(TKey key)
{
    // Make sure the hash code is positive.
    return m_comparer.GetHashCode(key) & 0x7fffffff;
}
```

**Visual Basic**

```
Private Function CalculateHashCode(ByVal key As TKey) As Integer
    ' Make sure the hash code is positive.
    Return m_comparer.GetHashCode(key) And &HFFFFFFF
End Function
```

With the *IsEmpty* property, users can check whether the collection is empty before doing some operations.

**C#**

```
/// <summary>
/// States if the AssociativeArrayHT(TKey,TValue) is empty.
/// </summary>
public bool IsEmpty
{
    get { return m_count <= 0; }
}
```

**Visual Basic**

```
''' <summary>
''' States if the AssociativeArrayHT(TKey,TValue) empty.
''' </summary>
Public ReadOnly Property IsEmpty() As Boolean
    Get
        Return m_count <= 0
    End Get
End Property
```

Users can also get the list of keys and values in the collection by calling the *Keys* and *Values* properties respectively.

**C#**

```
/// <summary>
/// Gets an array of current keys.
/// </summary>
public TKey[] Keys
{
    get
    {
        int index = 0;
        TKey[] keys = new TKey[Count];

```

```

        int bucketIndex = NULL_REFERENCE;
        int entryIndex = NULL_REFERENCE;
        while(MoveNext(ref bucketIndex, ref entryIndex))
        {
            keys[index++] = m_entries[entryIndex].Key;
        }

        return keys;
    }
}

/// <summary>
/// Gets an array of current values.
/// </summary>
public TValue[] Values
{
    get
    {
        TValue[] values = new TValue[Count];
        int index = 0;

        int bucketIndex = NULL_REFERENCE;
        int entryIndex = NULL_REFERENCE;
        while (MoveNext(ref bucketIndex, ref entryIndex))
        {
            values[index++] = m_entries[entryIndex].Value;
        }

        return values;
    }
}

```

### Visual Basic

```

''' <summary>
''' Gets an array of current keys.
''' </summary>
Public ReadOnly Property Keys() As TKey()
    Get
        Dim index As Integer = 0
        Dim rkeys As TKey() = New TKey(Count - 1) {}

        Dim bucketIndex As Integer = NULL_REFERENCE
        Dim entryIndex As Integer = NULL_REFERENCE
        While (MoveNext(bucketIndex, entryIndex))
            rkeys(index) = m_entries(entryIndex).Key
            index += 1
        End While

        Return rkeys
    End Get
End Property

''' <summary>
''' Gets an array of current values.
''' </summary>

```

```
Public ReadOnly Property Values() As TValue()
    Get
        Dim rvalues As TValue() = New TValue(Count - 1) {}
        Dim index As Integer = 0

        Dim bucketIndex As Integer = NULL_REFERENCE
        Dim entryIndex As Integer = NULL_REFERENCE
        While (MoveNext(bucketIndex, entryIndex))
            rvalues(index) = m_entries(entryIndex).Value
            index += 1
        End While

        Return rvalues
    End Get
End Property
```

Both properties traverse using the *MoveNext* method described in the following code. This method provides the users with a snapshot of the current *Keys* and *Values*. For now, this is just a placeholder. In Chapter 6, you learn how to return an enumerator for the *Keys* and *Values* properties that will stay in sync with the collection.

The *MoveNext* method is used for traversing the collection and is described as follows.

### C#

```
/// <summary>
/// Traverses the entries from the beginning to the end.
/// </summary>
/// <param name="bucketIndex">
/// The index of the bucket to start at or NULL_REFERENCE to start from the beginning.
/// </param>
/// <param name="entryIndex">
/// The index of the entry to start at or NULL_REFERENCE to start from the beginning.
/// </param>
/// <returns>False if the end of the bucket is reached, false otherwise.</returns>
internal bool MoveNext(ref int bucketIndex, ref int entryIndex)
{
    if (entryIndex == NULL_REFERENCE)
    {
        ++bucketIndex;

        // Check to see if we have reached the end of bucket array
        if (bucketIndex > m_buckets.Length)
        {
            return false;
        }

        // Keep checking buckets until we find one that isn't empty.
        while (m_buckets[bucketIndex] == NULL_REFERENCE)
        {

            // Check to see if we have reached the end of bucket array
            ++bucketIndex;
            if (bucketIndex >= m_buckets.Length)
            {
```

```

        return false;
    }

}

// Set the entryIndex to the first entry in the nonempty bucket
entryIndex = m_buckets[bucketIndex];
}

else
{
    entryIndex = m_entries[entryIndex].Next;

    // Recursively call this method if we have moved past the end of the current bucket
    if (entryIndex == NULL_REFERENCE)
    {
        return MoveNext(ref bucketIndex, ref entryIndex);
    }
}

return true;
}

```

### Visual Basic

```

''' <summary>
''' Traverses the entries from the beginning to the end.
''' </summary>
''' <param name="bucketIndex">
''' The index of the bucket to start at or NULL_REFERENCE to start from the beginning.
''' </param>
''' <param name="entryIndex">
''' The index of the entry to start at or NULL_REFERENCE to start from the beginning.
''' </param>
''' <returns>False if the end of the bucket is reached, false otherwise.</returns>
Friend Function MoveNext(ByRef bucketIndex As Integer, ByRef entryIndex As Integer) _
As Boolean
If (entryIndex = NULL_REFERENCE) Then
    bucketIndex += 1

    ' Check to see if we have reached the end of bucket array
    If (bucketIndex > m_buckets.Length) Then
        Return False
    End If

    ' Keep checking bucket until we find one that isn't empty.
    While (m_buckets(bucketIndex) = NULL_REFERENCE)

        ' Check to see if we have reached the end of bucket array
        bucketIndex += 1
        If (bucketIndex >= m_buckets.Length) Then
            Return False
        End If
    End While

    ' Set the entryIndex to the first entry in the nonempty bucket
    entryIndex = m_buckets(bucketIndex)

```

```

    Else
        entryIndex = m_entries(entryIndex).Next

        ' Recursively call this method if we have moved past the end of the current bucket
        If (entryIndex = NULL_REFERENCE) Then
            Return MoveNext(bucketIndex, entryIndex)
        End If
    End If

    Return True
End Function

```

The *MoveNext* method traverses the nodes by incrementing the values passed in through *bucketIndex* and *entryIndex*. If both values are *NULL\_REFERENCE*, it is assumed that the caller is starting from the beginning.

Sometimes users may want to know whether a particular key or value is present in the collection. The *ContainsKey* and *ContainsValue* methods can be used to do so.

### C#

```

/// <summary>
/// Checks to see if the specified key is present in the AssociativeArrayHT(TKey,TValue).
/// </summary>
/// <param name="key">The key to look for.</param>
/// <returns>True if the key was found, false otherwise.</returns>
public bool ContainsKey(TKey key)
{
    return !(FindKey(key).IsEmpty);
}

/// <summary>
/// Checks to see if the AssociativeArrayHT(TKey,TValue) contains the specified value.
/// </summary>
/// <param name="value">The value to look for.</param>
/// <returns>True if the value was found, false otherwise.</returns>
public bool ContainsValue(TValue value)
{
    return !(FindValue(value).IsEmpty);
}

```

### Visual Basic

```

''' <summary>
''' Checks to see if the specified key is present in the AssociativeArrayHT(TKey,TValue).
''' </summary>
''' <param name="key">The key to look for.</param>
''' <returns>True if the key was found, false otherwise.</returns>
Public Function ContainsKey(ByVal key As TKey) As Boolean
    Return Not (FindKey(key).IsEmpty)
End Function

```

```

''' <summary>
''' Checks to see if the AssociativeArrayHT(TKey,TValue) contains the specified value.
''' </summary>
''' <param name="value">The value to look for.</param>
''' <returns>True if the value was found, false otherwise.</returns>
Public Function ContainsValue(ByVal value As TValue) As Boolean
    Return Not (FindValue(value).IsEmpty)
End Function

```

The *ContainsValue* and *ContainsKey* methods use the *FindValue* and *FindKey* methods to tell whether an item is in the collection. They are defined as follows.

### C#

```

/// <summary>
/// Finds the entry that contains the specified key.
/// </summary>
/// <param name="key">The key to look for.</param>
/// <returns>The entry that contains the specified key, otherwise EntryData.EMPTY</returns>
EntryData FindKey(TKey key)
{
    if (IsEmpty)
    {
        return EntryData.EMPTY;
    }

    // Call out hashing function.
    int uHashcode = CalculateHashCode(key);

    // Calculate which bucket the key belongs in by doing a mod operation
    int bucketIndex = HashFunction(uHashcode);

    // Store the previous index in case the item is removed
    int previous = NULL_REFERENCE;
    for (int entryIndex = m_buckets[bucketIndex]; entryIndex != NULL_REFERENCE;
        entryIndex = m_entries[entryIndex].Next)
    {
        // Check to see if the hash code matches before doing a more complex search.
        if (m_entries[entryIndex].HashCode != uHashcode)
        {
            previous = entryIndex;
            continue;
        }

        if (m_comparer.Equals(m_entries[entryIndex].Key, key))
        {
            return new EntryData(this, entryIndex, previous, bucketIndex);
        }
        previous = entryIndex;
    }

    return EntryData.EMPTY;
}

```

```
/// <summary>
/// Finds the entry that contains the specified value.
/// </summary>
/// <param name="value">The value to look for.</param>
/// <returns>
/// The first entry that contains the specified value, otherwise EntryData.EMPTY.
/// </returns>
EntryData FindValue(TValue value)
{
    if (IsEmpty)
    {
        return EntryData.EMPTY;
    }

    EqualityComparer<TValue> comparer = EqualityComparer<TValue>.Default;

    for (int bucketIndex = 0; bucketIndex < m_buckets.Length; ++bucketIndex)
    {
        if (m_buckets[bucketIndex] == NULL_REFERENCE)
        {
            continue;
        }

        // Store the previous index in case the item is removed
        int previous = NULL_REFERENCE;
        for (int entryIndex = m_buckets[bucketIndex]; entryIndex != NULL_REFERENCE;
            entryIndex = m_entries[entryIndex].Next)
        {
            if (comparer.Equals(m_entries[entryIndex].Value, value))
            {
                return new EntryData(this, entryIndex, previous, bucketIndex);
            }

            previous = entryIndex;
        }
    }

    return EntryData.EMPTY;
}
```

### Visual Basic

```
''' <summary>
''' Finds the entry that contains the specified key.
''' </summary>
''' <param name="key">The key to look for.</param>
''' <returns>The entry that contains the specified key, otherwise EntryData.EMPTY</returns>
Private Function FindKey(ByVal key As TKey) As EntryData
    If (IsEmpty) Then
        Return EntryData.EMPTY
    End If
```

```

' Call out hashing function.
Dim uHashCode As Integer = CalculateHashCode(key)

' Calculate which bucket the key belongs in by doing a mod operation
Dim bucketIndex As Integer = HashFunction(uHashCode)

' Store the previous index in case the item is removed
Dim previous As Integer = NULL_REFERENCE
Dim entryIndex As Integer = m_buckets(bucketIndex)
While (entryIndex <> NULL_REFERENCE)
    ' Check to see if the hash code matches before doing a more complex search.
    If (m_entries(entryIndex).HashCode <> uHashCode) Then
        previous = entryIndex
        entryIndex = m_entries(entryIndex).Next
        Continue While
    End If

    If (m_comparer.Equals(m_entries(entryIndex).Key, key)) Then
        Return New EntryData(Me, entryIndex, previous, bucketIndex)
    End If
    previous = entryIndex
End While

Return EntryData.EMPTY
End Function

''' <summary>
''' Finds the entry that contains the specified value.
''' </summary>
''' <param name="value">The value to look for.</param>
''' <returns>
''' The first entry that contains the specified value, otherwise EntryData.EMPTY.
''' </returns>
Private Function FindValue(ByVal value As TValue) As EntryData
    If (IsEmpty) Then
        Return EntryData.EMPTY
    End If

    Dim comparer As EqualityComparer(Of TValue) = EqualityComparer(Of TValue).Default

    For bucketIndex As Integer = 0 To m_buckets.Length - 1
        If (m_buckets(bucketIndex) = NULL_REFERENCE) Then
            Continue For
        End If

        ' Store the previous index in case the item is removed
        Dim previous As Integer = NULL_REFERENCE
        Dim entryIndex As Integer = m_buckets(bucketIndex)
        While (entryIndex <> NULL_REFERENCE)
            If (comparer.Equals(m_entries(entryIndex).Value, value)) Then
                Return New EntryData(Me, entryIndex, previous, bucketIndex)
            End If
        End While
    Next
End Function

```

```
    previous = entryIndex
    entryIndex = m_entries(entryIndex).Next
End While
Next

Return EntryData.EMPTY
End Function
```

*FindValue* traverses the collection from the first bucket to the last bucket. The method checks each node by using a comparer to see whether the specified key or value is present at the current node.

Users need the ability to retrieve a value from the collection by using a key. The *Item* property allows them to do so.

### C#

```
/// <summary>
/// Gets or sets the value at the specified key.
/// </summary>
/// <param name="key">The key to use for finding the value.</param>
/// <returns>The value associated with the specified key.</returns>
public TValue this[TKey key]
{
    get
    {
        EntryData entry = FindKey(key);

        if (entry.IsEmpty)
        {
            throw new KeyNotFoundException("The specified key couldn't be located");
        }

        return entry.Value;
    }
}
```

### Visual Basic

```
''' <summary>
''' Gets or sets the value at the specified key.
''' </summary>
''' <param name="key">The key to use for finding the value.</param>
''' <returns>The value associated with the specified key.</returns>
Default Public Property Item(ByVal key As TKey) As TValue
    Get
        Dim entry As EntryData = FindKey(key)

        If (entry.IsEmpty) Then
            Throw New KeyNotFoundException("The specified key couldn't be located")
        End If

        Return entry.Value
    End Get
End Property
```

The *Item* property uses the *FindKey* method to locate the node that contains the specified key. If the node doesn't exist, the *Item* property throws an exception. This may not always be what users want. With the *TryGetValue* method, users can try to retrieve a value without throwing an exception if the key doesn't exist in the collection.

### C#

```
/// <summary>
/// Tries to get the value at the specified key without throwing an exception.
/// </summary>
/// <param name="key">The key to get the value for.</param>
/// <param name="value">The value that is associated with the specified key.</param>
/// <returns>True if the key was found, false otherwise.</returns>
public bool TryGetValue(TKey key, out TValue value)
{
    EntryData entry = FindKey(key);

    if (!entry.IsEmpty)
    {
        value = entry.Value;
        return true;
    }

    value = default(TValue);
    return false;
}
```

### Visual Basic

```
''' <summary>
''' Tries to get the value at the specified key without throwing an exception.
''' </summary>
''' <param name="key">The key to get the value for.</param>
''' <param name="value">The value that is associated with the specified key.</param>
''' <returns>True if the key was found, false otherwise.</returns>
Public Function TryGetValue(ByVal key As TKey, ByRef value As TValue) As Boolean
    Dim entry As EntryData = FindKey(key)

    If (Not entry.IsEmpty) Then
        value = entry.Value
        Return True
    End If

    value = CType(Nothing, TValue)
    Return False
End Function
```

## Using the Associative Array Class

Your boss wants you to implement a simple command line–driven phone book application. Users should be able to enter phone numbers and search for someone’s phone number. He doesn’t like it when people do not follow instructions, so he wants you to assume they will enter the data correctly. For the complete code, refer to the *Lesson3A* method in the Samples\Chapter 2\CS\Driver\Program.cs file for Microsoft Visual C# or the Samples\Chapter 2\VB\Driver\Module1.vb file for Microsoft Visual Basic.

The application runs in a continuous loop until the user selects the menu option to exit. It uses the following code in the *Lesson3A* method.

```
C#
static void Lesson3A()
{
    AssociativeArrayHT<string,string> phoneBook =
        new AssociativeArrayHT<string,string>(StringComparer.CurrentCultureIgnoreCase);
    string line;
    for (; ; )
    {
        PrintMenu();

        line = Console.ReadLine();
        line = line.Trim();
        if (string.IsNullOrEmpty(line))
        {
            continue;
        }
        switch (line[0])
        {
            case '1':
                AddNumber(phoneBook);
                break;
            case '2':
                LookupNumber(phoneBook);
                break;
            case '3':
                Show(phoneBook);
                break;
            case '4':
                return;
        }
    }
}
```

### Visual Basic

```
Sub Lesson3A()
    Dim phoneBook As AssociativeArrayHT(Of String, String) = _
        New AssociativeArrayHT(Of String, String)(StringComparer.CurrentCultureIgnoreCase)
    Dim line As String
    While (True)
        PrintMenu()

        line = Console.ReadLine()
        line = line.Trim()
        If (String.IsNullOrEmpty(line)) Then
            Continue While
        End If
        Select (line(0))
        Case "1"c
            AddNumber(phoneBook)
        Case "2"c
            LookupNumber(phoneBook)
        Case "3"c
            Show(phoneBook)
        Case "4"c
            Return
        End Select

    End While
End Sub
```

The following method is called at the beginning of the loop to display the menu options to the user.

### C#

```
static void PrintMenu()
{
    Console.WriteLine("1. Add Name");
    Console.WriteLine("2. Lookup Number");
    Console.WriteLine("3. Show Phonebook");
    Console.WriteLine("4. Exit");
    Console.Write("> ");
}
```

### Visual Basic

```
Sub PrintMenu()
    Console.WriteLine("1. Add Name")
    Console.WriteLine("2. Lookup Number")
    Console.WriteLine("3. Show Phonebook")
    Console.WriteLine("4. Exit")
    Console.Write("> ")
End Sub
```

Option 1 allows the user to add a person to the phone book by calling the following method.

**C#**

```
static void AddNumber(AssociativeArrayHT<string, string> phoneBook)
{
    Console.WriteLine("Enter the name to add");
    Console.Write("> ");
    string name = Console.ReadLine();

    if (string.IsNullOrEmpty(name))
    {
        return;
    }

    Console.WriteLine("Enter their phone number");
    Console.Write("> ");
    string number = Console.ReadLine();

    if (string.IsNullOrEmpty(number))
    {
        return;
    }

    phoneBook[name] = number;

    Console.WriteLine("{0}' phone number is {1}", name, phoneBook[name]);
}
```

**Visual Basic**

```
Sub AddNumber(ByVal phoneBook As AssociativeArrayHT(Of String, String))
    Console.WriteLine("Enter the name to add")
    Console.Write("> ")
    Dim name As String = Console.ReadLine()

    If (String.IsNullOrEmpty(name)) Then
        Return
    End If

    Console.WriteLine("Enter their phone number")
    Console.Write("> ")
    Dim number As String = Console.ReadLine()

    If (String.IsNullOrEmpty(number)) Then
        Return
    End If

    phoneBook(name) = number

    Console.WriteLine("{0}' phone number is {1}", name, phoneBook(name))
End Sub
```

### Output

```
1. Add Name
2. Lookup Number
3. Show Phonebook
4. Exit
> 1
Enter the name to add
> Akin, Cigdem
Enter their phone number
> (901) 555-0166
'Akin, Cigdem' phone number is (901) 555-0166
1. Add Name
2. Lookup Number
3. Show Phonebook
4. Exit
> 1
Enter the name to add
> Zimprich, Karin
Enter their phone number
> (901) 555-0177
'Zimprich, Karin' phone number is (901) 555-0177
1. Add Name
2. Lookup Number
3. Show Phonebook
4. Exit
> 1
Enter the name to add
> Zhang, Larry
Enter their phone number
> (901) 555-0133
'Zhang, Larry' phone number is (901) 555-0133
```

The method asks the user for the name of the person and phone number to associate with that person by using console input and output. The name is then added or overwritten using *phoneBook[name] = number*.

Option 2 allows the user to look up a person in the phone book by calling the following method.

### C#

```
static void LookupNumber(AssociativeArrayHT<string, string> phoneBook)
{
    Console.WriteLine("Enter name to lookup");
    Console.Write("> ");
    string name = Console.ReadLine();

    if (string.IsNullOrEmpty(name))
    {
        return;
    }
```

```
if (phoneBook.ContainsKey(name))
{
    Console.WriteLine("{0}' phone number is {1}", name, phoneBook[name]);
}
else
{
    Console.WriteLine("Couldn't find '{0}'", name);
}
}
```

### Visual Basic

```
Sub LookupNumber(ByVal phoneBook As AssociativeArrayHT(Of String, String))
    Console.WriteLine("Enter name to lookup")
    Console.Write("> ")
    Dim name As String = Console.ReadLine()

    If (String.IsNullOrEmpty(name)) Then
        Return
    End If

    If (phoneBook.ContainsKey(name)) Then
        Console.WriteLine("{0}' phone number is {1}", name, phoneBook(name))
    Else
        Console.WriteLine("Couldn't find '{0}'", name)
    End If
End Sub
```

### Output

```
1. Add Name
2. Lookup Number
3. Show Phonebook
4. Exit
> 2
Enter name to lookup
> Zimprich, Karin
'Zimprich, Karin' phone number is (901) 555-0177
1. Add Name
2. Lookup Number
3. Show Phonebook
4. Exit
> 2
Enter name to lookup
> Zakardissnehf, Kerstin
Couldn't find 'Zakardissnehf, Kerstin'
```

The method asks for the name of the person the user is looking up by using console input and output. The name is then checked to see if it is present in the phone book by calling `phoneBook.ContainsKey(name)`. If the name is present in the phone book, `phoneBook[name]` is then used to retrieve it.

Option 3 allows the user to display all numbers in the phone book.

#### C#

```
static void Show(AssociativeArrayHT<string, string> phoneBook)
{
    string[] names = phoneBook.Keys;

    for (int i = 0; i < names.Length; ++i)
    {
        Console.WriteLine("{0}' phone number is {1}", names[i], phoneBook[names[i]]);
    }
}
```

#### Visual Basic

```
Sub Show(ByVal phoneBook As AssociativeArrayHT(Of String, String))
    Dim names As String() = phoneBook.Keys

    For i As Integer = 0 To names.Length - 1
        Console.WriteLine("{0}' phone number is {1}", names(i), phoneBook(names(i)))
    Next
End Sub
```

#### Output

```
1. Add Name
2. Lookup Number
3. Show Phonebook
4. Exit
> 3
'Zhang, Larry' phone number is (901) 555-0133
'Zimprich, Karin' phone number is (901) 555-0177
'Akin, Cigdem' phone number is (901) 555-0166
```

Names are displayed by retrieving an array of keys from the phone book. Each key is then traversed and displayed on the screen along with its value.

Option 4 allows the user to exit.

## Summary

In this chapter, you learned how to use and implement associative arrays. You also learned that associative arrays have many different names among developers and different programming languages. They might be called maps, dictionaries, hash tables, and so on in different languages. Whatever the language may call them, associative arrays are useful when you have data that needs to be accessed with a key. Different implementations give you different performance gains and benefits.

## Chapter 3

# Understanding Collections: Queues, Stacks, and Circular Buffers

After completing this chapter, you will be able to

- Identify queues, stacks, and circular buffers.
- Design and implement queues, stacks, and circular buffers.
- Understand when and when not to use queues, stacks, and circular buffers.

## Queue Overview

A *queue* is a First In, First Out (FIFO) collection. A *FIFO* is a collection that resembles people waiting in a straight line. The first person to enter the line is the first person to leave the line. When the first people leave the line, they have been *dequeued*, or *popped off the front of the queue*, and as additional people join the line, they are *enqueued*, or *pushed onto the back of the queue*.

## Uses of Queues

Queues are normally used when items need to be processed in the order they are received. For example, say you have a thread that reads an ordered list of instructions from a file and another thread that processes those instructions. The reader thread needs to hold the information that it reads until the processing thread processes the items in the order they were read. A queue is the perfect collection for holding the data. The reader thread pushes the information onto the queue and the processing thread pops it off the queue.

## Queue Implementation

A queue may be implemented using a linked list or an array for the internal data storage. For information about when to use either a linked list or an array, see the advantages and disadvantages sections for each type in Chapter 1, “Understanding Collections: Arrays and Linked Lists.”

## Using an Array to Implement a Queue

To illustrate how to create a queue by using an array, you will add a class called *QueuedArray(T)*.



**Note** The Microsoft .NET Framework contains a class called *Queue(T)* that can be used for queues. The implementation uses an array for its internal data. The following code shows you how to implement from scratch a queue that uses an array, and later, one that uses a linked list for its internal storage.

### C#

```
[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(ArrayDebugView))]
public class QueuedArray<T>
{
    // Fields
    private const int GROW_BY = 10;
    private int m_count;
    private T[] m_data;
    private int m_head;
    private int m_tail;
    private int m_updateCode;

    // Constructors
    public QueuedArray();
    public QueuedArray(IEnumerable<T> items);
    public QueuedArray(int capacity);

    // Methods
    public void Clear();
    public bool Contains(T item);
    private void Initialize(int capacity);
    public T Peek();
    public T Pop();
    public void Push(T item);
    public T[] ToArray();

    // Properties
    public int Capacity { get; set; }
    public int Count { get; }
    public bool IsEmpty { get; }
}
```

### Visual Basic

```
<DebuggerTypeProxy(GetType(ArrayDebugView))>
<DebuggerDisplay("Count={Count}> _>
Public Class QueuedArray(Of T)
    ' Fields
    Private Const GROW_BY As Integer = 10
    Private m_count As Integer
    Private m_data As T()
```

```
Private m_head As Integer
Private m_tail As Integer
Private m_updateCode as Integer

' Constructors
Public Sub New()
Public Sub New(ByVal items As IEnumerable(Of T))
Public Sub New(ByVal capacity As Integer)

' Methods
Public Sub Clear()
Public Function Contains(ByVal item As T) As Boolean
Public Sub Initialize(ByVal capacity As Integer)
Public Function Peek() As T
Public Function Pop() As T
Public Sub Push(ByVal item As T)
Public Function ToArray() As T()

' Properties
Public Property Capacity As Integer
Public ReadOnly Property Count As Integer
Public ReadOnly Property IsEmpty As Boolean
End Class
```

The *m\_data* field is used as our internal data storage. The internal data storage is implemented as an array.

The *m\_updateCode* field is incremented each time the user modifies the list. The *m\_updateCode* field is used in Chapter 6, ".NET Collection Interfaces," to determine whether the collection has changed while the user is iterating over it. It is easier to add it to the code now instead of changing the code in Chapter 6.

## Creating Constructors

The *QueuedArray(T)* class contains three constructors. One constructor is for creating an empty class, the next one is for creating a class with default values, and the last one is used to create a queue with a specified internal array size. These constructors call the *Initialize* method to create the internal data storage object.

### C#

```
void Initialize(int capacity)
{
    m_data = new T[capacity];
}
```

### Visual Basic

```
Sub Initialize(ByVal capacity As Integer)
    m_data = New T(capacity - 1) {}
End Sub
```

The default constructor creates an empty object.

**C#**

```
/// <summary>
/// Initializes a new instance of the QueuedArray(T) class.
/// </summary>
public QueuedArray()
{
    Initialize(GROW_BY);
}
```

**Visual Basic**

```
''' <summary>
''' Initializes a new instance of the QueuedArray(T) class.
''' </summary>
Public Sub New()
    Initialize(GROW_BY)
End Sub
```

The following constructor creates a queue and adds the specified items to the end of the queue.

**C#**

```
/// <summary>
/// Initializes a new instance of the QueuedArray(T) class.
/// </summary>
public QueuedArray(IEnumerable<T> items)
{
    Initialize(GROW_BY);
    foreach (T item in items)
    {
        Push(item);
    }
}
```

**Visual Basic**

```
''' <summary>
''' Initializes a new instance of the QueuedArray(T) class.
''' </summary>
Public Sub New(ByVal items As IEnumerable(Of T))
    Initialize(GROW_BY)
    For Each item As T In items
        Push(item)
    Next
End Sub
```

The items are added to the queue by traversing the specified items and adding them to the queue.

The next constructor allows users to specify the initial size of the internal array.

**C#**

```
/// <summary>
/// Initializes a new instance of the QueuedArray(T) class
/// that is empty and has the specified initial capacity.
/// </summary>
/// <param name="capacity">
/// The number of elements that the new array can initially store.
/// </param>
public QueuedArray(int capacity)
{
    Initialize(capacity);
}
```

**Visual Basic**

```
''' <summary>
''' Initializes a new instance of the QueuedArray(T) class
''' that is empty and has the specified initial capacity.
''' </summary>
''' <param name="capacity">
''' The number of elements that the new array can initially store.
''' </param>
Public Sub New(ByVal capacity As Integer)
    Initialize(capacity)
End Sub
```

## Allowing Users to Add Items

Adding items to a queue is called *pushing items onto the queue*. This can be done using an *Enqueue* or *Push* method. The following implementation shows how to implement a *Push* method.

**C#**

```
/// <summary>
/// Adds the item to the end of the QueuedArray(T).
/// </summary>
/// <param name="item">The item to add to the end of the QueuedArray(T).</param>
public void Push(T item)
{
    if (m_count >= m_data.Length)
    {
        Capacity += GROW_BY;
    }

    m_data[m_tail] = item;
    m_tail = (m_tail + 1) % m_data.Length;
    ++m_count;
    ++m_updateCode;
}
```

### Visual Basic

```
'<summary>
''' Adds the item to the end of the QueuedArray(T).
'</summary>
''' <param name="item">The item to add to the end of the QueuedArray(T).</param>
Public Sub Push(ByVal item As T)
    If (m_count >= m_data.Length) Then
        Capacity += GROW_BY
    End If

    m_data(m_tail) = item
    m_tail = (m_tail + 1) Mod m_data.Length
    m_count += 1
    m_updateCode += 1
End Sub
```

The *Push* method adds the specified item to the end of the queue.

### Allowing Users to Remove Items

Removing items from the queue is called *popping items off the queue*. This can be done using a *Dequeue* or *Pop* method. The following implementation shows how to implement a *Pop* method.

#### C#

```
/// <summary>
/// Removes the first item from the QueuedArray(T).
/// </summary>
/// <returns></returns>
public T Pop()
{
    if (IsEmpty)
    {
        throw new InvalidOperationException("You cannot pop from an empty queue.");
    }

    T retval = m_data[m_head];

    m_head = (m_head + 1) % m_data.Length;

    --m_count;
    ++m_updateCode;

    return retval;
}
```

### Visual Basic

```
'<summary>
''' Removes the first item from the QueuedArray(T).
'</summary>
''' <returns></returns>
Public Function Pop() As T
```

```
If (IsEmpty) Then
    Throw New InvalidOperationException("You cannot pop from an empty queue.")
End If

Dim retval As T = m_data(m_head)

m_head = (m_head + 1) Mod m_data.Length

m_count -= 1
m_updateCode += 1

Return retval
End Function
```

The *Pop* method removes the first item from the queue and returns it to the user.

It is also useful to remove all items from the queue without repeatedly calling the *Pop* method. This is accomplished by calling the following method.

#### C#

```
/// <summary>
/// Removes all items from the QueuedArray(T).
/// </summary>
public void Clear()
{
    m_count = 0;
    m_head = 0;
    m_tail = 0;
    ++m_updateCode;
}
```

#### Visual Basic

```
''' <summary>
''' Removes all items from the QueuedArray(T).
''' </summary>
Public Sub Clear()
    m_count = 0
    m_head = 0
    m_tail = 0
    m_updateCode += 1
End Sub
```

## Adding Helper Methods and Properties

Users might want the ability to check the status of the queue. The *Contains* and *Peek* methods allow them to look at the contents of the queue, and the *Count* and *IsEmpty* properties allow them to look at the status of the queue.

The *Contains* method can be used to check to see whether an item is present in the queue. This may be useful if you are waiting for something to appear in the queue before you start popping things off it.

**C#**

```

/// <summary>
/// Checks if the specified data is present in the QueuedLinkedList(T).
/// </summary>
/// <param name="data">The data to look for.</param>
/// <returns>True if the data is found, false otherwise.</returns>
public bool Contains(T item)
{
    EqualityComparer<T> comparer = EqualityComparer<T>.Default;
    for (int i = 0; i < m_count; i++)
    {
        if (comparer.Equals(m_data[i], item))
        {
            return true;
        }
    }
    return false;
}

```

**Visual Basic**

```

''' <summary>
''' Checks if the specified data is present in the QueuedLinkedList(T).
''' </summary>
''' <param name="item">The data to look for.</param>
''' <returns>True if the data is found, false otherwise.</returns>
Public Function Contains(ByVal item As T) As Boolean
    Dim comparer As EqualityComparer(Of T) = EqualityComparer(Of T).Default
    For i As Integer = 0 To m_count - 1
        If (comparer.Equals(m_data(i), item)) Then
            Return True
        End If
    Next
    Return False
End Function

```

The *Contains* method walks each item in the queue and uses a comparer to see whether the items are equal.

Users may want to see what the first item in the queue is without actually removing it from the queue. With the following method, they can do that by looking at the *Head* element of the internal data storage.

**C#**

```

/// <summary>
/// Gets the first item in the QueuedArray(T) without removing it.
/// </summary>
/// <returns>The item at the front of the QueuedArray(T)</returns>
public T Peek()
{
    if (IsEmpty)

```

```
{  
    throw new InvalidOperationException("You cannot peek at an empty queue.");  
}  
  
return m_data[m_head];  
}
```

### Visual Basic

```
''' <summary>  
''' Gets the first item in the QueuedArray(T) without removing it.  
''' </summary>  
''' <returns>The item at the front of the QueuedArray(T)</returns>  
Public Function Peek() As T  
    If (IsEmpty) Then  
        Throw New InvalidOperationException("You cannot peek at an empty queue.")  
    End If  
  
    Return m_data(m_head)  
End Function
```

For some operations, the user needs to know whether the queue is empty. The following method provides this information.

### C#

```
/// <summary>  
/// States if the QueuedArray(T) is empty.  
/// </summary>  
public bool IsEmpty  
{  
    get { return m_count == 0; }  
}
```

### Visual Basic

```
''' <summary>  
''' States if the QueuedArray(T) is empty.  
''' </summary>  
Public ReadOnly Property IsEmpty() As Boolean  
    Get  
        Return m_count = 0  
    End Get  
End Property
```

The following property can be used to get the number of items in the queue.

### C#

```
/// <summary>  
/// Gets the number of elements actually contained in the QueuedArray(T).  
/// </summary>  
public int Count  
{  
    get { return m_count; }  
}
```

### Visual Basic

```

''' <summary>
''' Gets the number of elements actually contained in the QueuedArray(T).
''' </summary>
Public ReadOnly Property Count() As Integer
    Get
        Return m_count
    End Get
End Property

```

Finally, the following helper function copies the contents of the internal array into a new array. The new array is useful when code requires a basic array rather than an instance of the *QueueArray(T)* class.

### C#

```

/// <summary>
/// Copies the elements of the QueuedArray(T) to a new array.
/// </summary>
/// <returns>An array containing copies of the elements of the QueuedArray(T).</returns>
public T[] ToArray()
{
    T[] retval = new T[m_count];

    if (IsEmpty)
    {
        return retval;
    }

    if (m_head < m_tail)
    {
        Array.Copy(m_data, m_head, retval, 0, m_count);
    }
    else
    {
        Array.Copy(m_data, m_head, retval, 0, m_data.Length - m_head);
        Array.Copy(m_data, 0, retval, m_data.Length - m_head, m_tail);
    }

    return retval;
}

```

### Visual Basic

```

''' <summary>
''' Copies the elements of the QueuedArray(T) to a new array.
''' </summary>
''' <returns>An array containing copies of the elements of the QueuedArray(T).</returns>
Public Function ToArray() As T()
    Dim retval As New T(m_count - 1) {}

    If (IsEmpty) Then
        Return retval
    End If

```

```
If (m_head < m_tail) Then
    Array.Copy(m_data, m_head, retval, 0, m_count)
Else
    Array.Copy(m_data, m_head, retval, 0, m_data.Length - m_head)
    Array.Copy(m_data, 0, retval, m_data.Length - m_head, m_tail)
End If

Return retval
End Function
```

## Using a Linked List to Implement a Queue

To illustrate using a linked list to implement a queue, you will add a class called *QueuedLinkedList(T)*.

### C#

```
[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(ArrayDebugView))]
public class QueuedLinkedList<T>
{
    // Fields
    private DoubleLinkedList<T> m_data;

    // Constructors
    public QueuedLinkedList();
    public QueuedLinkedList(IEnumerable<T> items);

    // Methods
    public void Clear();
    public bool Contains(T item);
    public T Peek();
    public T Pop();
    public void Push(T item);
    public T[] ToArray();

    // Properties
    public int Count { get; }
    public bool IsEmpty { get; }
}
```

### Visual Basic

```
<DebuggerTypeProxy(GetType(ArrayDebugView)),>
<DebuggerDisplay("Count={Count}")> _
Public Class QueuedLinkedList(Of T)
    ' Fields
    Private m_data As DoubleLinkedList(Of T)

    ' Constructors
    Public Sub New()
    Public Sub New(ByVal items As IEnumerable(Of T))
```

```

' Methods
Public Sub Clear()
Public Function Contains(ByVal item As T) As Boolean
Public Function Peek() As T
Public Function Pop() As T
Public Sub Push(ByVal item As T)
Public Function ToArray() As T()

' Properties
Public ReadOnly Property Count As Integer
Public ReadOnly Property IsEmpty As Boolean
End Class

```

The *m\_data* field is used as the internal data storage. The internal data storage is implemented using the doubly linked list that was defined in Chapter 1. Each time the user calls the *Push* method, you pass the specified item to the *AddToEnd* method of the linked list. The doubly and singly linked lists you implemented in Chapter 1 store the tail in the field *m\_tail*. So in this type of scenario, the doubly and singly linked lists perform the same except for the additional memory that the doubly linked list stores for the previous reference. You could just as easily make the internal storage a singly linked list if you like.

The *m\_updateCode* from *m\_data* is used in Chapter 6 to determine whether the collection has changed while the user is iterating over it.

## Creating Constructors

The *QueuedArray(T)* class contains two constructors. One constructor is for creating an empty class and the other is for creating a class with default values.

The default constructor creates an empty object.

### C#

```

/// <summary>
/// Initializes a new instance of the QueuedLinkedList(T) class.
/// </summary>
public QueuedLinkedList()
{
    m_data = new DoubleLinkedList<T>();
}

```

### Visual Basic

```

''' <summary>
''' Initializes a new instance of the QueuedLinkedList(T) class.
''' </summary>
Public Sub New()
    m_data = New DoubleLinkedList(Of T)()
End Sub

```

The following constructor creates a queue and adds the specified items to the end of the queue.

#### C#

```
/// <summary>
/// Initializes a new instance of the QueuedLinkedList(T) class.
/// </summary>
public QueuedLinkedList(IEnumerable<T> items)
{
    m_data = new DoubleLinkedList<T>(items);
}
```

#### Visual Basic

```
''' <summary>
''' Initializes a new instance of the QueuedLinkedList(T) class.
''' </summary>
Public Sub New(ByVal items As IEnumerable(Of T))
    m_data = New DoubleLinkedList(Of T)(items)
End Sub
```

The items are added to the queue by creating the internal linked list with the specified items.

## Allowing Users to Add Items

Adding items to a queue is called *pushing items onto the queue*. This can be done using an *Enqueue* or a *Push* method. The following implementation shows how to implement the *Push* method.

#### C#

```
/// <summary>
/// Adds the item to the end of the QueuedLinkedList(T).
/// </summary>
/// <param name="item">The item to add to the end of the QueuedLinkedList(T).</param>
public void Push(T item)
{
    m_data.AddToEnd(item);
}
```

#### Visual Basic

```
''' <summary>
''' Adds the item to the end of the QueuedLinkedList(T).
''' </summary>
''' <param name="item">The item to add to the end of the QueuedLinkedList(T).</param>
Public Sub Push(ByVal item As T)
    m_data.AddToEnd(item)
End Sub
```

The *Push* method adds the specified item to the end of the queue.

## Allowing Users to Remove Items

Removing items from the queue is called *popping items off the queue*. This can be done using a *Dequeue* or *Pop* method. The following implementation shows how to implement the *Pop* method.

### C#

```
/// <summary>
/// Removes the first item from the QueuedLinkedList(T).
/// </summary>
/// <returns></returns>
public T Pop()
{
    if (IsEmpty)
    {
        throw new InvalidOperationException("You cannot pop from an empty queue.");
    }

    T retval = m_data.Head.Data;

    m_data.Remove(m_data.Head);

    return retval;
}
```

### Visual Basic

```
''> <summary>
''' Removes the first item from the QueuedLinkedList(T).
''' </summary>
''' <returns></returns>
Public Function Pop() As T
    If (IsEmpty) Then
        Throw New InvalidOperationException("You cannot pop from an empty queue.")
    End If

    Dim retval As T = m_data.Head.Data

    m_data.Remove(m_data.Head)

    Return retval
End Function
```

The *Pop* method removes the first item from the queue and returns it to the user.

It is also useful to remove all items from the queue without repeatedly calling the *Pop* method. This can be accomplished by calling the following method.

**C#**

```
/// <summary>
/// Removes all items from the QueuedLinkedList(T).
/// </summary>
public void Clear()
{
    m_data.Clear();
}
```

**Visual Basic**

```
'<summary>
'<summary> Removes all items from the QueuedLinkedList(T).
'</summary>
Public Sub Clear()
    m_data.Clear()
End Sub
```

The *Clear* method calls the *Clear* method of the internal data storage.

## Adding Helper Methods and Properties

Users might want the ability to check the status of the queue. The *Contains* and *Peek* methods allow them to look at the contents of the queue, and the *Count* and *IsEmpty* properties allow them to look at the status of the queue.

The *Contains* method can be used to check to see whether an item is present in the queue. This may be useful if you are waiting for something to appear in the queue before you start popping things off it.

**C#**

```
/// <summary>
/// Checks if the specified data is present in the QueuedLinkedList(T).
/// </summary>
/// <param name="data">The data to look for.</param>
/// <returns>True if the data is found, false otherwise.</returns>
public bool Contains(T item)
{
    return m_data.Contains(item);
}
```

**Visual Basic**

```
'<summary>
'<summary> Checks if the specified data is present in the QueuedLinkedList(T).
'</summary>
'<param name="item">The data to look for.</param>
'<returns>True if the data is found, false otherwise.</returns>
Public Function Contains(ByVal item As T) As Boolean
    Return m_data.Contains(item)
End Function
```

Users may want to see what the first item in a queue is without actually removing it. The following method allows them to do that by looking at the *Head* element of the internal data storage.

### C#

```
/// <summary>
/// Gets the first item in the QueuedLinkedList(T) without removing it.
/// </summary>
/// <returns>The item at the front of the QueuedLinkedList(T)</returns>
public T Peek()
{
    if (IsEmpty)
    {
        throw new InvalidOperationException("You cannot peek at an empty queue.");
    }

    return m_data.Head.Data;
}
```

### Visual Basic

```
''' <summary>
''' Gets the first item in the QueuedLinkedList(T) without removing it.
''' </summary>
''' <returns>The item at the front of the QueuedLinkedList(T)</returns>
Public Function Peek() As T
    If (IsEmpty) Then
        Throw New InvalidOperationException("You cannot peek at an empty queue.")
    End If

    Return m_data.Head.Data
End Function
```

For some operations, the user needs to know whether the queue is empty. The following property provides this information.

### C#

```
/// <summary>
/// States if the QueuedLinkedList(T) is empty.
/// </summary>
public bool IsEmpty
{
    get { return m_data.IsEmpty; }
}
```

### Visual Basic

```
''' <summary>
''' States if the QueuedLinkedList(T) is empty.
''' </summary>
Public ReadOnly Property IsEmpty() As Boolean
    Get
        Return m_data.IsEmpty
    End Get
End Property
```

The following property can be used to get the number of items in the queue.

**C#**

```
/// <summary>
/// Gets the number of elements actually contained in the QueuedLinkedList(T).
/// </summary>
public int Count
{
    get { return m_data.Count; }
}
```

**Visual Basic**

```
''' <summary>
''' Gets the number of elements actually contained in the QueuedLinkedList(T).
''' </summary>
Public ReadOnly Property Count() As Integer
    Get
        Return m_data.Count
    End Get
End Property
```

Finally, the following helper function copies the contents of the internal linked list into a new array. The new array is useful when code requires a basic array rather than an instance of the *QueuedLinkedList(T)* class.

**C#**

```
/// <summary>
/// Copies the elements of the QueuedLinkedList(T) to a new array.
/// </summary>
/// <returns>
/// An array containing copies of the elements of the QueuedLinkedList(T).
/// </returns>
public T[] ToArray()
{
    return m_data.ToArray();
}
```

**Visual Basic**

```
''' <summary>
''' Copies the elements of the QueuedLinkedList(T) to a new array.
''' </summary>
''' <returns>
''' An array containing copies of the elements of the QueuedLinkedList(T).
''' </returns>
Public Function ToArray() As T()
    Return m_data.ToArray()
End Function
```

## Using the Queue Classes

The following code walks you through using the *QueueArray(T)* and *ArrayEx(T)* classes to create a song request line simulator. Requests need to be stored until the current request finishes playing. Requests need to be played in the order they were received and played from beginning to end. A *QueueArray(T)* is used to hold the requests because it is a FIFO collection. The code does not use any threads because multithreading is not discussed until Chapter 8, “Using Threads with Collections.” Using collections in Windows Forms, Windows Presentation Foundation (WPF), and Microsoft Silverlight are discussed in Chapter 10, “Using Collections with Windows Forms,” and Chapter 11, “Using Collections in WPF and Silverlight,” so a console application is used for the user interface (UI). Because this is a chapter on collections, the code for writing to the console and other code that doesn’t pertain to using the *QueueArray(T)* is not discussed.



**More Info** For the complete code, refer to the *Lesson4A* method in the Samples\Chapter 3\CS\Driver\Program.cs file for Microsoft Visual C# or in the Samples\Chapter 3\VB\Driver\Module1.vb file for Microsoft Visual Basic.

You can begin with the variable declarations and the clearing of the console.

### C#

```
Request nowPlaying = null;
QueueArray<Request> requests = new QueueArray<Request>();
TimeSpan nextCallIn = TimeSpan.Zero;
TimeSpan currentTime = TimeSpan.Zero;
TimeSpan step = TimeSpan.FromSeconds(1);
Random rnd = new Random();
double playRate = 100;
int nextRow = 0;
int playRow = Console.WindowHeight - 2;
ArrayEx<Song> availableSongs = new ArrayEx<Song>(20);

Console.Clear();
```

### Visual Basic

```
Dim nowPlaying As Request = Nothing
Dim requests As Queue(Of Request) = New Queue(Of Request)()
Dim nextCallIn As TimeSpan = TimeSpan.Zero
Dim currentTime As TimeSpan = TimeSpan.Zero
Dim [step] As TimeSpan = TimeSpan.FromSeconds(1)
Dim rnd As Random = New Random()
Dim playRate As Double = 100
Dim nextRow As Integer = 0
Dim playRow As Integer = Console.WindowHeight - 2
Dim availableSongs As ArrayEx(Of Song) = New ArrayEx(Of Song)(20)

Console.Clear()
```

A random collection of songs is then created for the caller to request.

**C#**

```
// Create a random list of songs
for (int i = 0; i < 20; ++i)
{
    availableSongs.Add(new Song()
    {
        Name = string.Format("Song #{0}", i + 1),
        // Each song is from 3 minutes to 4 minutes
        Duration = TimeSpan.FromSeconds(3 * 60 + rnd.Next(121))
    });
}
```

**Visual Basic**

```
' Create a random list of songs
For i As Integer = 0 To 19
    availableSongs.Add(New Song() With _
    {
        .Name = String.Format("Song #{0}", i + 1),
        .Duration = TimeSpan.FromSeconds(3 * 60 + rnd.Next(120))
    })
Next
```

Next, a separator is drawn on the screen to separate the request from the now playing section.

**C#**

```
WriteSeparator(Console.WindowHeight - 3);
```

**Visual Basic**

```
WriteSeparator(Console.WindowHeight - 3)
```

The code then enters a loop until you press a key on the keyboard.

**C#**

```
while (!Console.KeyAvailable)
{
    if (nowPlaying != null)
    {
        // Check to see if the current request is finished
        if (currentTime >= nowPlaying.FinishTime)
        {
            nowPlaying = null;
        }
    }
    else
    {
        // Check to see if there are any requests waiting
        if (requests.Count > 0)
```

```
{  
    nowPlaying = requests.Pop();  
    nowPlaying.StartTime = currentTime;  
}  
}  
  
if (nowPlaying != null)  
{  
    WriteCurrentPlaying(nowPlaying, playRow, currentTime);  
}  
  
if (currentTime >= nextCallIn)  
{  
    // Pretend that someone has called in a request  
    Request request = new Request()  
    {  
        RequestTime = currentTime,  
        RequestSong = availableSongs[rnd.Next(availableSongs.Count - 1)]  
    };  
  
    WriteNextRequest(request, ref nextRow);  
  
    if (nextRow >= Console.WindowHeight - 2)  
    {  
  
        playRow = Math.Min(Console.CursorTop + 1, Console.BufferHeight - 2);  
  
        if (playRow == Console.BufferHeight - 2)  
        {  
            // Shifts all of the text up  
            AdvanceLine();  
  
            ++nextRow;  
        }  
  
        WriteSeparator(playRow - 1);  
    }  
  
    // Add the next request to the queue  
    requests.Push(request);  
  
    // Simulate someone calling in from 0 to 5 minutes from now  
    nextCallIn = currentTime + TimeSpan.FromSeconds(rnd.Next(5 * 60));  
}  
  
// Simulate actual time  
System.Threading.Thread.Sleep(Math.Max(0, (int)(step.TotalMilliseconds / playRate)));  
  
currentTime += step;  
}
```

### Visual Basic

```
While (Not Console.KeyAvailable)
    If (nowPlaying IsNot Nothing) Then
        ' Check to see if the current request is finished
        If (currentTime >= nowPlaying.FinishTime) Then
            nowPlaying = Nothing
        End If
    Else
        ' Check to see if there are any requests waiting
        If (requests.Count > 0) Then
            nowPlaying = requests.Pop()
            nowPlaying.StartTime = currentTime
        End If
    End If

    If (nowPlaying IsNot Nothing) Then
        WriteCurrentPlaying(nowPlaying, playRow, currentTime)
    End If

    If (currentTime >= nextCallIn) Then
        ' Pretend that someone has called in a request
        Dim request As Request = New Request() With _
        {
            .RequestTime = currentTime,
            .RequestSong = availableSongs(rnd.Next(availableSongs.Count - 1))
        }

        WriteNextRequest(request, nextRow)

        If (nextRow >= Console.WindowHeight - 2) Then
            playRow = Math.Min(Console.CursorTop + 1, Console.BufferHeight - 2)

            If (playRow = Console.BufferHeight - 2) Then
                ' Shifts all of the text up
                AdvanceLine()

                nextRow += 1
            End If

            WriteSeparator(playRow - 1)
        End If

        ' Add the next request to the queue
        requests.Push(request)

        ' Simulate someone calling in from 0 to 5 minutes from now
        nextCallIn = currentTime + TimeSpan.FromSeconds(rnd.Next(5 * 60))
    End If

    ' Simulate actual time
    System.Threading.Thread.Sleep(Math.Max(0, CInt([step].TotalMilliseconds / playRate)))

    currentTime += [step]
End While
```

### Output

```
Song #2: User request at 00:00:00
Song #8: User request at 00:04:48
Song #10: User request at 00:09:03
Song #7: User request at 00:11:56
Song #18: User request at 00:16:11
Song #11: User request at 00:18:30
Song #7: User request at 00:21:02
Song #16: User request at 00:24:49
Song #2: User request at 00:25:30
Song #12: User request at 00:27:55
Song #12: User request at 00:28:57
Song #18: User request at 00:32:07
Song #16: User request at 00:34:16
Song #18: User request at 00:35:46
Song #12: User request at 00:40:35
Song #6: User request at 00:41:31
```

---

```
Song #12: Playing -00:04:06
\.....
```

The loop checks to see whether a song is currently playing. If a song is playing, the code checks to see whether it has ended; otherwise, the code checks to see whether another song is available to play by running the following.

### C#

```
if (requests.Count > 0)
{
    nowPlaying = requests.Dequeue();
    nowPlaying.StartTime = currentTime;
}
```

### Visual Basic

```
If (requests.Count > 0) Then
    nowPlaying = requests.Dequeue()
    nowPlaying.StartTime = currentTime
End If
```

The code removes the next available request from *requests* if it contains any items. The code then checks to see whether someone has called in. If so, the code adds a new request to the queue by doing the following.

### C#

```
requests.Enqueue(request);
```

### Visual Basic

```
requests.Enqueue(request)
```

The code then randomly picks the next call-in time, pauses for a period of time, and then restarts the loop.

## Stack Overview

A *stack* is a Last In, First Out (LIFO) collection. A *LIFO* is a collection that resembles a stack of plates. The last plate added is the first plate that is removed.

### Uses of Stacks

A stack is normally used when you want to process first the last item received. Stacks can be used for variables in a function. When a function is called, each local variable is pushed onto the stack. After the function returns, each variable is popped off the stack, leaving the variables for the current function at the top of the stack. They are useful in some types of parsing. If you are interested in seeing a parsing example, take a look at Reverse Polish notation (RPN).

## Stack Implementation

A stack can be implemented using a linked list or an array for the internal data storage. For information about when to use either a linked list or an array, see the advantages and disadvantages sections for each type in Chapter 1.

### Adding the Common Functionality

Both implementations have the same *IsEmpty*, *Count*, *Clear*, and *Contains* methods and properties. Each method or property simply calls the same method or property of the internal data storage object, so they will be shown here.



**Note** The .NET Framework contains a class called *Stack(T)* that can be used for stacks. The implementation uses an array for its internal data. The following code shows you how to implement from scratch a stack that uses an array and one that uses a linked list for its internal storage.

**C#**

```

/// <summary>
/// States if the StackedArray(T) is empty.
/// </summary>
public bool IsEmpty
{
    get { return m_data.IsEmpty; }
}

/// <summary>
/// Gets the number of elements in the StackedArray(T).
/// </summary>
public int Count
{
    get { return m_data.Count; }
}

/// <summary>
/// Removes all items from the StackedArray(T).
/// </summary>
public void Clear()
{
    m_data.Clear();
}

/// <summary>
/// Checks if the specified data is present in the StackedArray(T).
/// </summary>
/// <param name="data">The data to look for.</param>
/// <returns>True if the data is found, false otherwise.</returns>
public bool Contains(T item)
{
    return m_data.Contains(item);
}

```

**Visual Basic**

```

''' <summary>
''' States if the StackedArray(T) is empty.
''' </summary>
Public ReadOnly Property IsEmpty()
    Get
        Return m_data.IsEmpty
    End Get
End Property

''' <summary>
''' Gets the number of elements in the StackedArray(T).
''' </summary>
Public ReadOnly Property Count() As Integer
    Get
        Return m_data.Count
    End Get
End Property

```

```

''' <summary>
''' Removes all items from the StackedArray(T).
''' </summary>
Public Sub Clear()
    m_data.Clear()
End Sub

''' <summary>
''' Checks if the specified data is present in the StackedArray(T).
''' </summary>
''' <param name="item">The data to look for.</param>
''' <returns>True if the data is found, false otherwise.</returns>
Public Function Contains(ByVal item As T) As Boolean
    Return m_data.Contains(item)
End Function

```

## Using an Array to Implement a Stack

To illustrate how to use an array to implement a stack, you will create a class called *StackedArray(T)*.

### C#

```

[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(ArrayDebugView))]
public class StackedArray<T>
{
    // Fields
    private ArrayEx<T> m_data;

    // Constructors
    public StackedArray();
    public StackedArray(IEnumerable<T> items);

    // Methods
    public void Clear();
    public bool Contains(T item);
    public T Peek();
    public T Pop();
    public void Push(T item);
    public T[] ToArray();

    // Properties
    public int Count { get; }
    public bool IsEmpty { get; }
}

```

### Visual Basic

```

<DebuggerDisplay("Count={Count}")> _
<DebuggerTypeProxy(GetType(ArrayDebugView))> _
Public Class StackedArray(Of T)

    ' Fields
    Private m_data As ArrayEx(Of T)

```

```

' Constructors
Public Sub New()
Public Sub New(ByVal items As IEnumerable(Of T))

' Methods
Public Sub Clear()
Public Function Contains(ByVal item As T) As Boolean
Public Function Peek() As T
Public Function Pop() As T
Public Sub Push(ByVal item As T)
Public Function ToArray() As T()

' Properties
Public ReadOnly Property Count As Integer
Public ReadOnly Property IsEmpty As Object
End Class

```

The *m\_data* field is used for the internal data storage and implemented as an *ArrayEx(T)* class.

The *m\_updateCode* from *m\_data* is used in Chapter 6 to determine whether the collection has changed while the user is iterating over it.

## Creating Constructors

The *StackedArray(T)* class contains two constructors. One constructor is for creating an empty class and the other is for creating a class with default values.

The default constructor creates an empty object.

### C#

```

/// <summary>
/// Initializes a new instance of the StackedArray(T) class that is empty.
/// </summary>
public StackedArray()
{
    m_data = new ArrayEx<T>();
}

```

### Visual Basic

```

''' <summary>
''' Initializes a new instance of the StackedArray(T) class that is empty.
''' </summary>
Public Sub New()
    m_data = New ArrayEx(Of T)()
End Sub

```

The following constructor creates a stack and adds the specified items to the stack.

#### C#

```
/// <summary>
/// Initializes a new instance of the StackedArray(T) class with the specified items.
/// </summary>
/// <param name="items">The items to add to the stack.</param>
public StackedArray(IEnumerable<T> items)
{
    m_data = new ArrayEx<T>(items);
}
```

#### Visual Basic

```
''' <summary>
''' Initializes a new instance of the StackedArray(T) class with the specified items.
''' </summary>
''' <param name="items">The items to add to the stack.</param>
Public Sub New(ByVal items As IEnumerable(Of T))
    m_data = New ArrayEx(Of T)(items)
End Sub
```

The items are added to the stack by creating the internal data storage with the specified items.

## Allowing Users to Add to Items

Adding an item to the stack is called *pushing it onto the stack*. The following implementation shows how to implement the *Push* method.

#### C#

```
/// <summary>
/// Adds the item to the top of the StackedArray(T).
/// </summary>
/// <param name="item">The item to add to the top of the StackedArray(T).</param>
public void Push(T item)
{
    m_data.Add(item);
}
```

#### Visual Basic

```
''' <summary>
''' Adds the item to the top of the StackedArray(T).
''' </summary>
''' <param name="item">The item to add to the top of the StackedArray(T).</param>
Public Sub Push(ByVal item As T)
    m_data.Add(item)
End Sub
```

The method adds the specified item to the end of the array.

## Allowing Users to Remove Items

Removing an item from the stack is called *popping it off the stack*. The following implementation shows how to implement the *Pop* method.

### C#

```
/// <summary>
/// Removes the an item from the top of the StackedArray(T).
/// </summary>
/// <returns>The item at the top of the StackedArray(T).</returns>
public T Pop()
{
    if (IsEmpty)
    {
        throw new InvalidOperationException("You cannot pop from an empty stack.");
    }

    T retval = m_data[m_data.Count - 1];

    m_data.RemoveAt(m_data.Count - 1);

    return retval;
}
```

### Visual Basic

```
''' <summary>
''' Removes the an item from the top of the StackedArray(T).
''' </summary>
''' <returns>The item at the top of the StackedArray(T).</returns>
Public Function Pop() As T
    If (IsEmpty) Then
        Throw New InvalidOperationException("You cannot pop from an empty stack.")
    End If

    Dim retval As T = m_data(m_data.Count - 1)

    m_data.RemoveAt(m_data.Count - 1)

    Return retval
End Function
```

The method removes the item at the end of the array.

## Adding Helper Methods

Sometimes you might find it necessary to look at the item about to be removed from the stack without physically removing it. Unfortunately, this cannot be accomplished by using any of the methods described so far. You can achieve this with the *Peek* method, which is shown next.

**C#**

```
/// <summary>
/// Gets the item at the top of the StackedArray(T) without removing it.
/// </summary>
/// <returns>
/// The item at the top of the StackedArray(T)
/// </returns>
public T Peek()
{
    if (IsEmpty)
    {
        throw new InvalidOperationException("You cannot peek at an empty stack.");
    }

    return m_data[m_data.Count - 1];
}
```

**Visual Basic**

```
''' <summary>
''' Gets the item at the top of the StackedArray(T) without removing it.
''' </summary>
''' <returns>The item at the top of the StackedArray(T)</returns>
Public Function Peek() As T

    If (IsEmpty) Then
        Throw New InvalidOperationException("You cannot peek at an empty stack.")
    End If

    Return m_data(m_data.Count - 1)
End Function
```

The *Peek* method returns the last item added to the stack.

The next method returns an array that contains all of the values in the stack. The items are added to the array in LIFO order. That is, the first item in the array would be the last one added to the stack.

**C#**

```
/// <summary>
/// Copies the elements of the StackedArray(T) to a new array.
/// </summary>
/// <returns>
/// An array containing copies of the elements of the StackedArray(T).
/// The data in the array will be in LIFO order.
/// </returns>
public T[] ToArray()
{
    T[] tmp = new T[Count];

    for (int i = 0; i < Count; ++i)
```

```

    {
        tmp[i] = m_data[Count - i - 1];
    }

    return tmp;
}

```

### Visual Basic

```

''' <summary>
''' Copies the elements of the StackedArray(T) to a new array.
''' </summary>
''' <returns>
''' An array containing copies of the elements of the StackedArray(T).
''' The data in the array will be in LIFO order.
''' </returns>
Public Function ToArray() As T()
    Dim tmp As T() = New T(Count - 1) {}

    For i As Integer = 0 To Count - 1
        tmp(i) = m_data(Count - i - 1)
    Next

    Return tmp
End Function

```

## Using a Linked List to Implement a Stack

To illustrate using a linked list to implement a stack, you will create a class called *StackedLinkedList(T)*.

### C#

```

[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(ArrayDebugView))]
public class StackedLinkedList<T>
{
    // Fields
    private DoubleLinkedList<T> m_data;

    // Methods
    public StackedLinkedList();
    public StackedLinkedList(IEnumerable<T> items);
    public void Clear();
    public bool Contains(T item);
    public T Peek();
    public T Pop();
    public void Push(T item);
    public T[] ToArray();

    // Properties
    public int Count { get; }
    public bool IsEmpty { get; }

}

```

### Visual Basic

```
<DebuggerDisplay("Count={Count}")> _  
Public Class StackedLinkedList(Of T)  
  
    ' Fields  
    Private m_data As DoubleLinkedList(Of T)  
  
    ' Constructors  
    Public Sub New()  
        Public Sub New(ByVal items As IEnumerable(Of T))  
  
    ' Methods  
    Public Sub Clear()  
    Public Function Contains(ByVal item As T) As Boolean  
    Public Function Peek() As T  
    Public Function Pop() As T  
    Public Sub Push(ByVal item As T)  
    Public Function ToArray() As T()  
  
    ' Properties  
    Public ReadOnly Property Count As Integer  
    Public ReadOnly Property IsEmpty As Boolean  
End Class
```

The *m\_data* field is used for the internal data storage and is implemented as a *DoubleLinkedList(T)*.

The *m\_updateCode* from *m\_data* is used in Chapter 6 to determine whether the collection has changed while the user is iterating over it.

### Creating Constructors

The *StackedLinkedList(T)* class contains two constructors. One constructor is for creating an empty class and the other is for creating a class with default values.

The default constructor creates an empty object.

### C#

```
/// <summary>  
/// Initializes a new instance of the StackedLinkedList(T) class that is empty.  
/// </summary>  
public StackedLinkedList()  
{  
    m_data = new DoubleLinkedList<T>();  
}
```

### Visual Basic

```
''' <summary>  
''' Initializes a new instance of the StackedLinkedList(T) class that is empty.  
''' </summary>  
Public Sub New()  
    m_data = New DoubleLinkedList(Of T)()  
End Sub
```

The following constructor creates a stack and adds the specified items to the stack.

#### C#

```
/// <summary>
/// Initializes a new instance of the StackedLinkedList(T) class with the specified items.
/// </summary>
/// <param name="items">The items to add to the stack.</param>
public StackedLinkedList(IEnumerable<T> items)
{
    m_data = new DoubleLinkedList<T>(items);
}
```

#### Visual Basic

```
''' <summary>
''' Initializes a new instance of the StackedLinkedList(T) class with the specified items.
''' </summary>
''' <param name="items">The items to add to the stack.</param>
Public Sub New(ByVal items As IEnumerable(Of T))
    m_data = New DoubleLinkedList(Of T)(items)
End Sub
```

The items are added to the stack by creating the internal data storage with the specified items.

## Allowing Users to Add Items

The following implementation shows how to implement the *Push* method to allow users to push an item on the stack.

#### C#

```
/// <summary>
/// Adds the item to the top of the StackedLinkedList(T).
/// </summary>
/// <param name="item">The item to add to the top of the StackedLinkedList(T).</param>
public void Push(T item)
{
    m_data.AddToEnd(item);
}
```

#### Visual Basic

```
''' <summary>
''' Adds the item to the top of the StackedLinkedList(T).
''' </summary>
''' <param name="item">The item to add to the top of the StackedLinkedList(T).</param>
Public Sub Push(ByVal item As T)
    m_data.AddToEnd(item)
End Sub
```

The method adds the specified item to the end of the linked list.

## Allowing Users to Remove Items

The following implementation shows how to implement the *Pop* method to allow users to pop an item off the stack.

### C#

```
/// <summary>
/// Removes the an item from the top of the StackedLinkedList(T).
/// </summary>
/// <returns>The item at the top of the StackedLinkedList(T).</returns>
public T Pop()
{
    if (IsEmpty)
    {
        throw new InvalidOperationException("You cannot pop from an empty stack.");
    }

    T retval = m_data.Tail.Data;

    m_data.Remove(m_data.Tail);

    return retval;
}
```

### Visual Basic

```
''' <summary>
''' Removes the an item from the top of the StackedLinkedList(T).
''' </summary>
''' <returns>The item at the top of the StackedLinkedList(T).</returns>
Public Function Pop() As T
    If (IsEmpty) Then
        Throw New InvalidOperationException("You cannot pop from an empty stack.")
    End If

    Dim retval As T = m_data.Tail.Data

    m_data.Remove(m_data.Tail)

    Return retval
End Function
```

The method removes the item at the end of the list.

## Adding Helper Methods

Sometimes you might find it necessary to look at the item about to be removed from the stack without physically removing it. Unfortunately, this cannot be accomplished by using any of the methods described so far. You can achieve this with the *Peek* method, which is shown next.

**C#**

```

/// <summary>
/// Gets the item at the top of the StackedLinkedList(T) without removing it.
/// </summary>
/// <returns>The item at the top of the StackedLinkedList(T)</returns>
public T Peek()
{
    if (IsEmpty)
    {
        throw new InvalidOperationException
            ("You cannot get the top item of an empty stack.");
    }

    return m_data.Tail.Data;
}

```

**Visual Basic**

```

''' <summary>
''' Gets the item at the top of the StackedLinkedList(T) without removing it.
''' </summary>
''' <returns>The item at the top of the StackedLinkedList(T)</returns>
Public Function Peek() As T
    If (IsEmpty) Then
        Throw New InvalidOperationException _
            ("You cannot get the top item of an empty stack.")
    End If

    Return m_data.Tail.Data
End Function

```

The *Peek* method will return the last item added to the list.

The next method returns an array that contains all of the values in the stack. The items are added to the array in LIFO order. That is, the first item in the array would be the last one added to the stack.

**C#**

```

/// <summary>
/// Copies the elements of the StackedLinkedList(T) to a new array.
/// </summary>
/// <returns>
/// An array containing copies of the elements of the StackedLinkedList(T).
/// The data in the array will be in LIFO order.
/// </returns>
public T[] ToArray()
{
    T[] tmp = new T[Count];

    int i = 0;
    for(DoubleLinkedListNode<T> curr = m_data.Tail; curr != null; curr = curr.Previous)

```

```
{  
    tmp[i++] = curr.Data;  
}  
  
return tmp;  
}
```

### Visual Basic

```
''' <summary>  
''' Copies the elements of the StackedLinkedList(T) to a new array.  
''' </summary>  
''' <returns>  
''' An array containing copies of the elements of the StackedLinkedList(T).  
''' The data in the array will be in LIFO order.  
''' </returns>  
Public Function ToArray() As T()  
    Dim tmp As T() = New T(Count - 1) {}  
  
    Dim i As Integer = 0  
    Dim curr As DoubleLinkedListNode(Of T) = m_data.Tail  
    While (curr IsNot Nothing)  
        tmp(i) = curr.Data  
        i = i + 1  
        curr = curr.Previous  
    End While  
  
    Return tmp  
End Function
```

## Using the Stack Classes

Your boss now wants you to simulate plates being used in a cafeteria. The simulation should include people standing in line to get a plate, sitting down and using the plate, and returning the plate to be cleaned.



**More Info** For the complete code, you can refer to the *Lesson6A* method in the Samples\Chapter 3\CS\Driver\Program.cs file for C# or the Samples\Chapter 3\VB\Driver\Module1.vb file for Visual Basic.

First, you need to define a *Plate* class.

### C#

```
class Plate  
{  
    public int Number;  
    public TimeSpan NextOperation;  
}
```

### Visual Basic

```
Class Plate
    Public Number As Integer
    Public NextOperation As TimeSpan
End Class
```

*Number* is a unique number assigned to each plate that you can use to track the plate as it goes through the simulation. *NextOperation* is used internally by the simulator to determine when the next operation should happen, such as cleaning the plate, using the plate, and so on.

Next, you need to create the variables and the initial stack of plates.

### C#

```
StackedArray<Plate> cleanPlates = new StackedArray<Plate>();
SingleLinkedList<Plate> usingPlates = new SingleLinkedList<Plate>();
QueuedArray<Plate> dirtyPlates = new QueuedArray<Plate>();
Random rnd = new Random();
TimeSpan currentTime = TimeSpan.Zero;
TimeSpan step = TimeSpan.FromSeconds(1);
double playRate = 100.0;
TimeSpan nextCustomer = TimeSpan.Zero;
bool hasWarned = false;

for (int i = 0; i < 20; ++i)
{
    cleanPlates.Push(new Plate() { Number = i });
}
```

### Visual Basic

```
Dim cleanPlates As New StackedArray(Of Plate)()
Dim usingPlates As New SingleLinkedList(Of Plate)()
Dim dirtyPlates As New QueuedArray(Of Plate)()
Dim rnd As New Random()
Dim currentTime As TimeSpan = TimeSpan.Zero
Dim [step] As TimeSpan = TimeSpan.FromSeconds(1)
Dim playRate As Double = 100.0
Dim nextCustomer As TimeSpan = TimeSpan.Zero
Dim hasWarned As Boolean = False

For i As Integer = 0 To 19
    cleanPlates.Push(New Plate() With {.Number = i})
Next
```

The variable *cleanPlates* represents the stack of clean plates that customers can get. The variable *usingPlates* represents the collection of plates that customers are using. The variable *dirtyPlates* represents the plates that are currently on the conveyer belt to be cleaned. The *for* statement adds 20 plates to the stack.

The application then loops until the user presses a key.

**C#**

```
while (!Console.KeyAvailable)
{
    // Simulates the person eating and carrying the plate to the conveyer belt to be cleaned
    . . .

    // Simulates plates being cleaned
    . . .

    // Simulates the person standing in line
    . . .

    // Simulates actual time
    System.Threading.Thread.Sleep(Math.Max(0, (int)(step.TotalMilliseconds / playRate)));

    currentTime += step;
}
```

**Visual Basic**

```
While (Not Console.KeyAvailable)
    ' Simulates the person eating and carrying the plate to the conveyer belt to be cleaned
    . . .

    ' Simulates plates being cleaned
    . . .

    ' Simulates the person standing in line
    . . .

    ' Simulates actual time
    System.Threading.Thread.Sleep(Math.Max(0, CInt([step].TotalMilliseconds / playRate)))

    currentTime += [step]
End While
```

The following code is used to simulate a person eating and carrying his or her plate to the conveyer belt to be cleaned.

**C#**

```
if (usingPlates.Count > 0)
{
    SingleLinkedListNode<Plate> node = usingPlates.Head;
    while (node != null)
    {
        if (currentTime >= node.Data.NextOperation)
        {
            Plate plate = node.Data;

            Console.WriteLine("Plate {0} is being taken to the cleaners", plate.Number);
```

```

        // Takes 20 seconds to clean a plate
        plate.NextOperation = currentTime + TimeSpan.FromSeconds(20);

        // Add the plate to the conveyer belt to be cleaned
        dirtyPlates.Push(plate);
        SingleLinkedListNode<Plate> tmp = node.Next;
        usingPlates.Remove(node);
        node = tmp;
    }
    else
    {
        node = node.Next;
    }
}
}

```

### Visual Basic

```

If (usingPlates.Count > 0) Then
    Dim node As SingleLinkedListNode(Of Plate) = usingPlates.Head
    While (node IsNot Nothing)
        If (currentTime >= node.Data.NextOperation) Then
            Dim plate As Plate = node.Data

            Console.WriteLine("Plate {0} is being taken to the cleaners", plate.Number)

            ' Takes 20 seconds to clean a plate
            plate.NextOperation = currentTime + TimeSpan.FromSeconds(20)

            ' Add the plate to the conveyer belt to be cleaned
            dirtyPlates.Push(plate)
            Dim tmp As SingleLinkedListNode(Of Plate) = node.Next
            usingPlates.Remove(node)
            node = tmp
        Else
            node = node.Next
        End If
    End While
End If

```

The *NextOperation* field is used for stating when the customer will finish eating. Customers do not finish eating in the order that they start, so a linked list is used to simulate the eating customers. The collection is traversed with each loop to check whether the customer has finished eating. After the customer has finished, the plate is then added to the *dirtyPlates* collection.

The following code is then used to simulate plates being cleaned.

```

C#
if (dirtyPlates.Count > 0)
{
    Plate plate = dirtyPlates.Peek();

    if (currentTime >= plate.NextOperation)

```

```
{  
    Console.WriteLine("Plate {0} has been cleaned", plate.Number);  
    dirtyPlates.Pop();  
    cleanPlates.Push(plate);  
}  
}
```

### Visual Basic

```
If (dirtyPlates.Count > 0) Then  
    Dim plate As Plate = dirtyPlates.Peek()  
  
    If (currentTime >= plate.NextOperation) Then  
        Console.WriteLine("Plate {0} has been cleaned", plate.Number)  
        dirtyPlates.Pop()  
        cleanPlates.Push(plate)  
    End If  
End If
```

The *NextOperation* field is used to state when a plate is finished being cleaned. Plates are cleaned in the order they are received, so a queue is used for the conveyer belt.

The following code simulates a customer standing in line and getting a plate.

### C#

```
If (currentTime >= nextCustomer) Then  
    If (Not cleanPlates.IsEmpty) Then  
        Dim plate As Plate = cleanPlates.Pop()  
  
        ' It can take up to 3 minutes for the customer to eat  
        plate.NextOperation = currentTime + TimeSpan.FromSeconds(rnd.Next(3 * 60))  
        usingPlates.AddToEnd(plate)  
  
        ' Next customer can take up to 30 seconds  
        nextCustomer = currentTime + TimeSpan.FromSeconds(rnd.Next(30))  
  
        Console.WriteLine("Plate {0} has been taken", plate.Number)  
  
        hasWarned = False  
    Else  
        If (Not hasWarned) Then  
            Console.WriteLine("Waiting for plates to be cleaned!")  
            hasWarned = True  
        End If  
    End If  
End If
```

### Visual Basic

```
if (currentTime >= nextCustomer)  
{  
    if (!cleanPlates.IsEmpty)
```

```
{  
    Plate plate = cleanPlates.Pop();  
  
    // It can take up to 3 minutes for the customer to eat  
    plate.NextOperation = currentTime + TimeSpan.FromSeconds(rnd.Next(3 * 60));  
    usingPlates.AddToEnd(plate);  
  
    // Next customer can take up to 30 seconds  
    nextCustomer = currentTime + TimeSpan.FromSeconds(rnd.Next(30));  
  
    Console.WriteLine("Plate {0} has been taken", plate.Number);  
  
    hasWarned = false;  
}  
else  
{  
    if (!hasWarned)  
    {  
        Console.WriteLine("Waiting for plates to be cleaned!");  
        hasWarned = true;  
    }  
}  
}  
}
```

The *NextOperation* field is used to state when a customer will finish eating. The next customer gets a plate off the stack and uses it. If no plates are present, the customer waits until one is available.

### Output

```
Plate 19 has been taken  
Plate 18 has been taken  
Plate 17 has been taken  
Plate 16 has been taken  
Plate 17 is being taken to the cleaners  
Plate 15 has been taken  
Plate 14 has been taken  
Plate 17 has been cleaned  
Plate 17 has been taken  
Plate 13 has been taken  
Plate 17 is being taken to the cleaners  
Plate 12 has been taken  
Plate 14 is being taken to the cleaners  
Plate 17 has been cleaned  
Plate 17 has been taken  
Plate 11 has been taken  
Plate 10 has been taken  
Plate 9 has been taken  
Plate 14 has been cleaned  
Plate 14 has been taken  
Plate 8 has been taken  
Plate 7 has been taken  
Plate 13 is being taken to the cleaners  
Plate 6 has been taken
```

```
Plate 15 is being taken to the cleaners
Plate 13 has been cleaned
Plate 13 has been taken
Plate 14 is being taken to the cleaners
Plate 19 is being taken to the cleaners
Plate 15 has been cleaned
Plate 15 has been taken
Plate 18 is being taken to the cleaners
Plate 14 has been cleaned
Plate 19 has been cleaned
Plate 16 is being taken to the cleaners
Plate 9 is being taken to the cleaners
Plate 19 has been taken
Plate 18 has been cleaned
Plate 18 has been taken
Plate 16 has been cleaned
Plate 9 has been cleaned
Plate 9 has been taken
Plate 10 is being taken to the cleaners
Plate 12 is being taken to the cleaners
Plate 16 has been taken
Plate 10 has been cleaned
Plate 9 is being taken to the cleaners
Plate 11 is being taken to the cleaners
Plate 8 is being taken to the cleaners
Plate 12 has been cleaned
```

## Circular Buffer Overview

A circular buffer is a fixed-size data structure that is arranged so that the end of the buffer connects to the beginning of the buffer.

## Uses of Circular Buffers

Circular buffers are useful when you need a FIFO collection that can be a fixed size. In general, the size of a circular buffer does not increase as you add and remove items from it. The data you are adding may cause an increase in memory but not in the circular buffer data structure itself. So you can have an endless FIFO that does not increase in memory because items are truncated or removed from the buffer when it is full.

Older data will be overwritten if the user attempts to add data to a full buffer. That functionality is helpful if, say for example, you have a large number of messages coming in that you need to process but old messages are not very important. If your buffer size is 1000, you will start to drop messages that are more than a 1000 messages old when the buffer fills up.

## Circular Buffer Implementation

Because a circular buffer is a fixed size collection, it is generally better to use an array as the internal structure instead of a linked list. If you plan on doing a lot of resizing, it may be better to use a linked list as the internal structure, but you should also consider collection types other than circular buffers as well.

### Getting Started

To illustrate using an array to create a circular buffer, you will create a class called *CircularBuffer(T)*.

#### C#

```
[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(ArrayDebugView))]
public class CircularBuffer<T>
{
    // Fields
    private int m_capacity;
    private int m_count;
    private T[] m_data;
    private int m_end;
    private FullOperations m_fullOperation;
    private int m_start;
    private int m_updateCode;

    // Methods
    public CircularBuffer(int size);
    public CircularBuffer(int size, IEnumerable<T> items);
    public void Clear();
    public bool Contains(T item);
    private void Initialize(int size);
    public T Peek();
    public T Pop();
    public void Push(T item);
    public T[] ToArray();

    // Properties
    public int Capacity { get; set; }
    public int Count { get; }
    public FullOperations FullOperation { get; set; }
    public bool IsEmpty { get; }
    public bool IsFull { get; }
}
```

#### Visual Basic

```
<DebuggerDisplay("Count={Count}") >
<DebuggerTypeProxy(GetType(ArrayDebugView))> 
Public Class CircularBuffer(Of T)
```

```

' Fields
Private m_capacity As Integer
Private m_count As Integer
Private m_data As T()
Private m_end As Integer
Private m_fullOperation As FullOperations
Private m_start As Integer
Private m_updateCode as Integer
' Constructors
Public Sub New(ByVal size As Integer)
Public Sub New(ByVal size As Integer, ByVal items As IEnumerable(Of T))

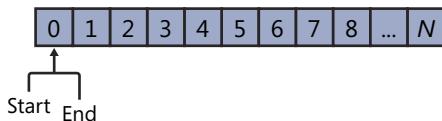
' Methods
Public Sub Clear()
Public Function Contains(ByVal item As T) As Boolean
Public Sub Initialize(ByVal size As Integer)
Public Function Peek() As T
Public Function Pop() As T
Public Sub Push(ByVal item As T)
Public Function ToArray() As T()

' Properties
Public Property Capacity As Integer
Public ReadOnly Property Count As Integer
Public Property FullOperation As FullOperations
Public ReadOnly Property IsEmpty As Boolean
Public ReadOnly Property IsFull As Boolean
End Class

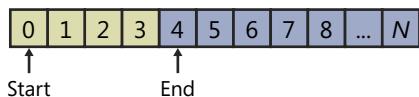
```

## Understanding the Internal Storage

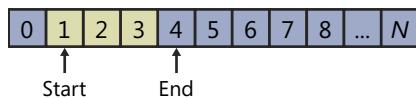
You can implement a circular buffer in many ways. Each way has its advantages and disadvantages. For this example, you will be using start and end pointers along with capacity and count variables. The start and end pointers help us keep track of where the data begins and ends in the internal array. With an empty buffer, both pointers point to the same element.



When you add a value, it is placed at the location of the end pointer. The end pointer is then incremented so that the process can be repeated on the next add. The lightly shaded area in the following illustration denotes a filled element.

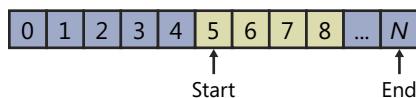


When you remove a value, it is removed from the start location. Start is then incremented so that the process can be repeated with the next removal.

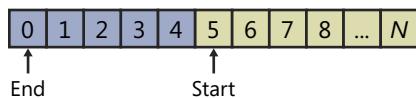


At some point, the start or end values will be at the end of the list. To make the collection a circular buffer, you move the pointer to the beginning of the array and start the remove or add process from there.

Here the end pointer is pointing to the end of the array before an add.



When that happens, end then points to the beginning of the array after the add.



You can accomplish this in two ways. You can increment the index and then check to see whether it has reached the capacity of the array, as shown in the following code.

```
C#
++index;
if (index >= capacity)
{
    index = 0;
}
```

#### Visual Basic

```
index += 1
If (index >= capacity) Then
    index = 0
End If
```

Or you can use the modulus (mod) operation to keep the value from 0 to (capacity – 1).

```
C#
index = (index + 1) % capacity;
```

#### Visual Basic

```
index = (index + 1) Mod capacity
```

For this case, you will be using the mod operation.

You may have noticed that the number of elements in the array can be calculated by a mathematical formula.

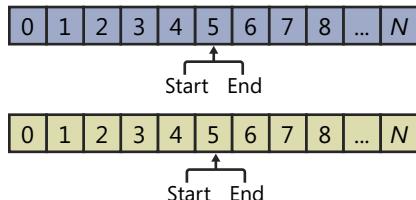
### C#

```
if (start < end)
{
    count = end - start;
}
else
{
    count = end + (capacity - start);
}
```

### Visual Basic

```
If (start < [end]) Then
    count = [end] - start
Else
    count = [end] + (capacity - start)
End If
```

What you may not have noticed yet is that there are two times when start and end can be equal to each other. Both pointers can be equal to each other when the buffer is full and when the buffer is empty.



To distinguish the two, you need an additional variable called a *count*. The count variable needs to be updated whenever a value is removed or added to the list. This variable also eliminates the calculation that is needed whenever the count is requested. Using the count variable, you can create three status properties for the user.

### C#

```
/// <summary>
/// Gets the number of elements actually contained in the CircularBuffer(T).
/// </summary>
public int Count
{
    get { return m_count; }
}

/// <summary>
/// States if the CircularBuffer(T) is empty.
/// </summary>
public bool IsEmpty
```

```

{
    get { return m_count <= 0; }
}

/// <summary>
/// States if the CircularBuffer(T) is full.
/// </summary>
public bool IsFull
{
    get { return m_count == m_capacity; }
}

```

### Visual Basic

```

''' <summary>
''' Gets the number of elements actually contained in the CircularBuffer(T).
''' </summary>
Public ReadOnly Property Count() As Integer
    Get
        Return m_count
    End Get
End Property

''' <summary>
''' States if the CircularBuffer(T) is empty.
''' </summary>
Public ReadOnly Property IsEmpty() As Boolean
    Get
        Return m_count <= 0
    End Get
End Property

''' <summary>
''' States if the CircularBuffer(T) is full.
''' </summary>
Public ReadOnly Property IsFull() As Boolean
    Get
        Return m_count = m_capacity
    End Get
End Property

```

## Creating Constructors

The *CircularBuffer(T)* class contains two constructors. One constructor is for creating an empty class and the other is for creating a class with default values. Both require the user to specify the size of the circular buffer.

Both constructors call the *Initialize* method so that duplicate code does not have to be maintained.

**C#**

```
void Initialize(int size)
{
    if (size <= 0)
    {
        throw new ArgumentOutOfRangeException("size");
    }
    m_data = new T[size];
    m_capacity = size;
    FullOperation = FullOperations.Ignore;
}
```

**Visual Basic**

```
Sub Initialize(ByVal size As Integer)
    If (size <= 0) Then
        Throw New ArgumentOutOfRangeException("size")
    End If
    m_data = New T(size - 1) {}
    m_capacity = size
    FullOperation = FullOperations.Ignore
End Sub
```

The method simply creates an array of the size passed in and sets the *FullOperation* property to a default operation that it performs when the user adds to a full circular buffer.

The default constructor creates an empty buffer of the specified size.

**C#**

```
/// <summary>
/// Initializes a new instance of the CircularBuffer(T) class
/// with the specified size as the capacity.
/// </summary>
/// <param name="size">The capacity of the buffer</param>
public CircularBuffer(int size)
{
    Initialize(size);
}
```

**Visual Basic**

```
''' <summary>
''' Initializes a new instance of the CircularBuffer(T) class
''' with the specified size as the capacity.
''' </summary>
''' <param name="size">The capacity of the buffer</param>
Public Sub New(ByVal size As Integer)
    Initialize(size)
End Sub
```

The following constructor creates a circular buffer of the specified size and adds the specified items to the end of the circular buffer.

## C#

```

/// <summary>
/// Initializes a new instance of the CircularBuffer(T) class
/// with the specified items and capacity.
/// </summary>
/// <param name="size">The capacity of the buffer</param>
/// <param name="items">The collection of items you want to add to the buffer.</param>
public CircularBuffer(int size, IEnumerable<T> items)
{
    Initialize(size);

    foreach (T item in items)
    {
        if (m_count >= size)
        {
            throw new ArgumentOutOfRangeException
                ("size",
                "The number of items in the collection is larger than the size");
        }
        m_data[m_end++] = item;
        ++m_count;
    }
}

```

## Visual Basic

```

''' <summary>
''' Initializes a new instance of the CircularBuffer(T) class
''' with the specified items and capacity.
''' </summary>
''' <param name="size">The capacity of the buffer</param>
''' <param name="items">The collection of items you want to add to the buffer.</param>
Public Sub New(ByVal size As Integer, ByVal items As IEnumerable(Of T))
    Initialize(size)

    For Each item As T In items
        If (m_count >= size) Then
            Throw New ArgumentOutOfRangeException _
                ("size", "The number of items in the collection is larger than the size")
        End If
        m_data(m_end) = item
        m_end += 1
        m_count += 1
    Next
End Sub

```

The constructor adds the specified items to the internal array by walking each item and adding it to the current end of the array.

## Allowing Users to Add Items

Users need the ability to add items to the buffer. The following method allows them to do this, but you should consider a few things when adding items to a full buffer. Reallocating a circular buffer to handle a new item is a costly operation, so when implementing a circular buffer, you must decide on what to do when the buffer becomes full. Normally, a user will want to overwrite old values in a circular buffer, but sometimes the user will want to throw an exception or ignore the newly added value. You may want to throw an exception if all values should be kept to inform the user that her buffer size is too small or that one of the threads is not keeping up. Sometimes it's more beneficial to keep older data rather than the newer, so you may want to ignore new values. The *CircularBuffer(T)* is designed to handle all three cases. First, there is an enumeration and a property to let the user specify what he or she wants to do.

C#

```
/// <summary>
/// Enumeration of what should happen if the circular buffer is full.
/// </summary>
public enum FullOperations
{
    /// <summary>
    /// The value being pushed should be ignored.
    /// </summary>
    Ignore,
    /// <summary>
    /// The first value should be popped off.
    /// </summary>
    Pop,
    /// <summary>
    /// An exception should be thrown.
    /// </summary>
    Error
}

/// <summary>
/// Gets or sets what to do when a user pushes to a full buffer.
/// </summary>
public FullOperations FullOperation
{
    get { return m_fullOperation; }
    set { m_fullOperation = value; }
}
```

**Visual Basic**

```

''' <summary>
''' Enumeration of what should happen if the circular buffer is full.
''' </summary>
Public Enum FullOperations
    ''' <summary>
    ''' The value being pushed should be ignored.
    ''' </summary>
    Ignore
    ''' <summary>
    ''' The first value should be popped off.
    ''' </summary>
    Pop
    ''' <summary>
    ''' An exception should be thrown.
    ''' </summary>
    [Error]
End Enum

''' <summary>
''' Gets or sets what to do when a user pushes to a full buffer.
''' </summary>
Public Property FullOperation() As FullOperations
    Get
        Return m_fullOperation
    End Get
    Set(ByVal value As FullOperations)
        m_fullOperation = value
    End Set
End Property

```

Next, the *Push* method adds an item to the buffer and does what the user requested on a full buffer.

**C#**

```

/// <summary>
/// Adds the item to the end of the CircularBuffer(T).
/// </summary>
/// <param name="item">The item to add to the end of the buffer.</param>
public void Push(T item)
{
    if (IsFull)
    {
        switch (m_fullOperation)
        {
            case FullOperations.Ignore:
                // Do not do anything
                return;
            case FullOperations.Pop:
                Pop();
                break;
            case FullOperations.Error:
            default:
                throw new InvalidOperationException("You cannot add to a full buffer");
        }
    }
}

```

```

        }
    }

    m_data[m_end] = item;
    ++m_count;
    ++m_updateCode;
    m_end = (m_end + 1) % m_capacity;
}

```

### Visual Basic

```

''' <summary>
''' Adds the item to the end of the CircularBuffer(T).
''' </summary>
''' <param name="item">The item to add to the end of the buffer.</param>
Public Sub Push(ByVal item As T)
    If (IsFull) Then
        Select Case (m_fullOperation)
            Case FullOperations.Ignore
                ' Do not do anything
                Exit Sub
            Case FullOperations.Pop
                Pop()
                Exit Select
            Case FullOperations.Error
                Throw New InvalidOperationException("You cannot add to a full buffer")
                Exit Select
            Case Else
                Throw New InvalidOperationException("You cannot add to a full buffer")
                Exit Select
        End Select
    End If

    m_data(m_end) = item
    m_count += 1
    m_updateCode += 1
    m_end = (m_end + 1) Mod m_capacity
End Sub

```

## Allowing Users to Remove Items

Users need the ability to remove items from the buffer in the order they were added. With the following method, users can pop items off the buffer.

### C#

```

/// <summary>
/// Removes the first item from the CircularBuffer(T).
/// </summary>
/// <returns>The first item in the CircularBuffer(T).</returns>
public T Pop()
{
    if (IsEmpty)

```

```

    {
        throw new InvalidOperationException
            ("You cannot remove an item from an empty collection.");
    }

    T value = m_data[m_start];
    m_start = (m_start + 1) % m_capacity;
    --m_count;
    ++m_updateCode;
    return value;
}

```

**Visual Basic**

```

''' <summary>
''' Removes the first item from the CircularBuffer(T).
''' </summary>
''' <returns>The first item in the CircularBuffer(T).</returns>
Public Function Pop() As T
    If (IsEmpty) Then
        Throw New InvalidOperationException _
            ("You cannot remove an item from an empty collection.")
    End If

    Dim value As T = m_data(m_start)
    m_start = (m_start + 1) Mod m_capacity
    m_count -= 1
    m_updateCode += 1
    Return value
End Function

```

With the *Clear* method, users can remove all items from the buffer without having to repeatedly call *Pop* until the buffer is empty.

**C#**

```

/// <summary>
/// Removes all items from the CircularBuffer(T).
/// </summary>
public void Clear()
{
    m_count = 0;
    m_start = 0;
    m_end = 0;
    ++m_updateCode;
}

```

**Visual Basic**

```

''' <summary>
''' Removes all items from the CircularBuffer(T).
''' </summary>
Public Sub Clear()
    m_count = 0
    m_start = 0
    m_end = 0
    m_updateCode += 1
End Sub

```

## Adding Helper Methods and Properties

Sometimes you might want to view the next element in the buffer without actually removing it. You can do so with the following method.

### C#

```
/// <summary>
/// Gets the item the first item in the CircularBuffer(T) without removing it.
/// </summary>
/// <returns>The first item in the CircularBuffer(T).</returns>
public T Peek()
{
    if (IsEmpty)
    {
        throw new InvalidOperationException
            ("You cannot view an item in an empty collection.");
    }

    return m_data[m_start];
}
```

### Visual Basic

```
''' <summary>
''' Gets the item the first item in the CircularBuffer(T) without removing it.
''' </summary>
''' <returns>The first item in the CircularBuffer(T).</returns>
Public Function Peek() As T

    If (IsEmpty) Then
        Throw New InvalidOperationException _
            ("You cannot view an item in an empty collection.")
    End If
    Return m_data(m_start)
End Function
```

With the following method, users can check and see if an item is present in the buffer. This may be useful if a user wants to interact only with a collection that contains a specific item.

### C#

```
/// <summary>
/// Checks if the specified data is present in the CircularBuffer(T).
/// </summary>
/// <param name="data">The data to look for.</param>
/// <returns>True if the data is found, false otherwise.</returns>
public bool Contains(T item)
{
    EqualityComparer<T> comparer = EqualityComparer<T>.Default;
    int i = m_start;
    int index = 0;
    while (index < m_count)
    {
        if (comparer.Equals(m_data[i], item))
```

```

    {
        return true;
    }
    i = (i + 1) % m_capacity;
    ++index;
}

return false;
}

```

### Visual Basic

```

''' <summary>
''' Checks if the specified data is present in the CircularBuffer(T).
''' </summary>
''' <param name="item">The data to look for.</param>
''' <returns>True if the data is found, false otherwise.</returns>
Public Function Contains(ByVal item As T) As Boolean
    Dim comparer As EqualityComparer(Of T) = EqualityComparer(Of T).Default
    Dim i As Integer = m_start
    Dim index As Integer = 0
    While (index < m_count)
        If (comparer.Equals(m_data(i), item)) Then
            Return True
        End If
        i = (i + 1) Mod m_capacity
        index += 1
    End While

    Return False
End Function

```

The following method copies the contents of the buffer into an array. This is useful when the user wants to interact with the data in array form.

### C#

```

/// <summary>
/// Copies the elements of the CircularBuffer(T) to a new array.
/// </summary>
/// <returns>An array containing copies of the elements of the CircularBuffer(T).</returns>
public T[] ToArray()
{
    T[] retval = new T[m_count];

    if (IsEmpty)
    {
        return retval;
    }

    if (m_start < m_end)
    {
        Array.Copy(m_data, m_start, retval, 0, m_count);
    }
    else

```

```
{  
    Array.Copy(m_data, m_start, retval, 0, m_capacity - m_start);  
    Array.Copy(m_data, 0, retval, m_capacity - m_start, m_end);  
}  
  
return retval;  
}
```

### Visual Basic

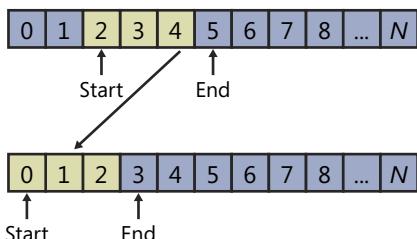
```
''' <summary>  
''' Copies the elements of the CircularBuffer(T) to a new array.  
''' </summary>  
''' <returns>An array containing copies of the elements of the CircularBuffer(T).</returns>  
Public Function ToArray() As T()  
    Dim retval As New T(m_count - 1) {}  
  
    If (IsEmpty) Then  
        Return retval  
    End If  
  
    If (m_start < m_end) Then  
        Array.Copy(m_data, m_start, retval, 0, m_count)  
    Else  
        Array.Copy(m_data, m_start, retval, 0, m_capacity - m_start)  
        Array.Copy(m_data, 0, retval, m_capacity - m_start, m_end)  
    End If  
  
    Return retval  
End Function
```

## Changing Capacity

If it is determined during run time that the initial buffer size is too small, the user may want to increase the size of the buffer. There are two cases that you need to be aware of when reallocating a buffer that is not empty.

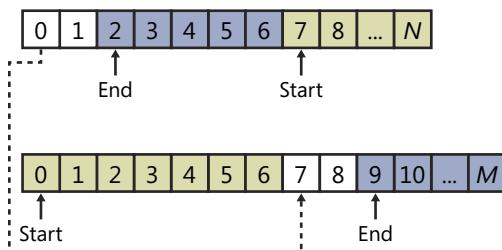
### When the Start Is Smaller Than the End

If the start pointer is smaller than the end pointer, you can allocate the new buffer and simply copy the old data to the beginning of the new array.



## When the Start Is Larger Than the End

If the start pointer is larger than the end pointer, you need to copy two sets of data. The values from the start pointer to the end of the array need to be copied to the front of the new array; the values from the start of the array to the end pointer need to be appended to the end of the new array data.



You need to create the following property for users to change the capacity of the buffer.

**C#**

```
/// <summary>
/// Gets or sets the capacity of the CircularBuffer(T).
/// </summary>
public int Capacity
{
    get { return m_capacity; }
    set
    {
        if (value < Count)
        {
            throw new NotSupportedException
                ("The capacity has to be larger than the current count.");
        }

        T[] tmp = new T[value];

        // We can just create a new buffer if it is empty.
        if (IsEmpty)
        {
            m_capacity = value;
            m_data = tmp;
            return;
        }

        // We can simply copy the data if the end hasn't wrapped around yet.
        if (m_start < m_end)
        {
            // We will need to copy the data from the old array to the new one
            // All data will be copied to the beginning of the new array
            Array.Copy(m_data, m_start, tmp, 0, m_count);
        }
    }
}
```

```
m_data = tmp;
m_capacity = value;
m_start = 0;
m_end = m_count;
return;
}

// First, we will copy all data from the start to the physical end of the buffer
Array.Copy(m_data, m_start, tmp, 0, m_data.Length - m_start);

// Next, we will copy the items from the physical start of the buffer to the end
Array.Copy(m_data, 0, tmp, m_data.Length - m_start, m_end);

m_data = tmp;
m_start = 0;
m_end = m_count;
m_capacity = value;
}
}
```

### Visual Basic

```
''' <summary>
''' Gets or sets the capacity of the CircularBuffer(T).
''' </summary>
Public Property Capacity() As Integer
    Get
        Return m_capacity
    End Get
    Set(ByVal value As Integer)
        If (value < Count) Then
            Throw New NotSupportedException _
                ("The capacity has to be larger than the current count.")
        End If

        Dim tmp As New T(value - 1) {}

        ' We can just create a new buffer if it is empty.
        If (IsEmpty) Then
            m_capacity = value
            m_data = tmp
            Exit Property
        End If

        ' We can simply copy the data if the end hasn't wrapped around yet.
        If (m_start < m_end) Then
            ' We will need to copy the data from the old array to the new one
            ' All data will be copied to the beginning of the new array
            Array.Copy(m_data, m_start, tmp, 0, m_count)
```

```

    m_data = tmp
    m_capacity = value
    m_start = 0
    m_end = m_count
    Exit Property
End If

' First, we will copy all data from the start to the physical end of the buffer
Array.Copy(m_data, m_start, tmp, 0, m_data.Length - m_start)

' Next, we will copy the items from the physical start of the buffer to the end
Array.Copy(m_data, 0, tmp, m_data.Length - m_start, m_end)

m_data = tmp
m_start = 0
m_end = m_count
m_capacity = value
End Set
End Property

```

## Using the *CircularBuffer(T)* Class

The following code walks you through using use the *CircularBuffer(T)* and *ArrayEx(T)* classes to create a song request line simulator. The song request line will be the same as the one in the “Using the Queue Classes” section earlier in this chapter, except it will use a *CircularBuffer(T)* instead of a *QueueArray(T)*.



**More Info** For the complete code, refer to the *Lesson5A* method in the Samples\Chapter 3\CS\Driver\Program.cs file for C# or the Samples\Chapter 3\VB\Driver\Module1.vb file for Visual Basic.

Note the following code from the *Lesson5A* method., which is modified from *Lesson4A*.

### C#

```
QueueArray<Request> requests = new QueueArray<Request>();
```

### Visual Basic

```
Dim requests As QueueArray(Of Request) = New QueueArray(Of Request)()
```

The preceding code changes to the following.

### C#

```
CircularBuffer<Request> requests = new CircularBuffer<Request>(10);
```

### Visual Basic

```
Dim requests As CircularBuffer(Of Request) = New CircularBuffer(Of Request)(10)
```

The constructor now takes an integer that states the size of the circular buffer. You might notice that there is a possibility that the request line that uses the circular buffer may get more requests than it can play. When using a queue, this could cause the application to eventually run out of memory. A circular buffer doesn't grow in size, so the buffer cannot become larger than the value specified in the constructor.

### Output

```
Song #2: User request at 00:00:00
Song #8: User request at 00:04:48
Song #10: User request at 00:09:03
Song #7: User request at 00:11:56
Song #18: User request at 00:16:11
Song #11: User request at 00:18:30
Song #7: User request at 00:21:02
Song #16: User request at 00:24:49
Song #2: User request at 00:25:30
Song #12: User request at 00:27:55
Song #12: User request at 00:28:57
Song #18: User request at 00:32:07
Song #16: User request at 00:34:16
Song #18: User request at 00:35:46
Song #12: User request at 00:40:35
Song #6: User request at 00:41:31
```

---

```
Song #12: Playing -00:04:06
\.....
```

## Summary

In this chapter, you saw how to use and implement queues, stacks, and circular buffers. You also saw that queues and circular buffers are FIFO collections and stacks are LIFO collections. Circular buffers are good for when you have a fixed size collection that needs to be accessed in a FIFO manner. Remember that adding to a full circular buffer requires a performance penalty if you resize the buffer or loss of data if you decide not to resize the buffer. Queues, on the other hand, should be used when the buffer size is not fixed. Of course, these are just general guidelines; you may sometimes find it useful to use a circular buffer with a collection that isn't a fixed size.



Part II

## .NET Built-in Collections



# Chapter 4

# Generic Collections

After completing this chapter, you will be able to

- Understand the equality and ordering comparers.
- Understand delegates, anonymous methods, and lambda expressions.
- Understand when and how to use the *List(T)* class.
- Understand when and how to use the *LinkedList(T)* class.

## Understanding the Equality and Ordering Comparers

It is important to know that some of the classes in the *System.Collections.Generic* namespace use an equality and ordering comparer in some of their methods. You can spend considerable precious time debugging code if you don't fully understand these comparers.

### Understanding the Equality Comparer

Some of the classes in the *System.Collections.Generic* namespace implement what is called a *default equality comparison*. This means that the method checks to see whether *T* implements the *IEquatable(T)* interface; if so, the method then calls the *Equals(T)* method exposed by that interface. If the interface isn't implemented, the method calls the *Object.Equals(Object)* method on *T* instead. The default implementation of *Object.Equals(Object)* checks for reference equality for reference types and bitwise equality for value types.

### How Reference Equality Works

Reference equality verifies that two variables reference the same object. Another way of saying this is that it verifies that two references point to the same *instance* of an object. Take a look at the following example.

```
C#
// Define an empty class named ClassA
class ClassA { }

// Create two new instances of ClassA
ClassA x = new ClassA();
ClassA y = new ClassA();

// Assign a reference to instance x in refx
ClassA refx = x;
```

**Visual Basic**

```
Public Class ClassA
```

```
End Class
```

```
Dim x = New ClassA()  
Dim y = New ClassA()
```

```
Dim refx = x
```

The following code would display *false* because *x* and *y* reference two different instances of the class *ClassA*.

**C#**

```
Console.WriteLine(x.Equals(y));
```

**Visual Basic**

```
Console.WriteLine(x.Equals(y))
```

The following code would display *true* because *x* and *refx* both reference the same instance of the class *ClassA*.

**C#**

```
Console.WriteLine(x.Equals(refx));
```

**Visual Basic**

```
Console.WriteLine(x.Equals(refx))
```

## How Bitwise Equality Works

Bitwise equality verifies that two objects have the same binary representation. Take a look at the following example.

**C#**

```
int x = 1;  
int y = 2;
```

```
Console.WriteLine(x.Equals(y));
```

**Visual Basic**

```
Dim x As Integer = 1  
Dim y As Integer = 2
```

```
Console.WriteLine(x.Equals(y))
```

The variables *x* and *y* are set to two different values, so the console would display *false*.

But the following two lines of code would both return *true* because *x* and *y* are now binary equivalents.

**C#**

```
int x = 1;
int y = 1;
Console.WriteLine(x.Equals(y));

y = x;
Console.WriteLine(x.Equals(y));
```

**Visual Basic**

```
Dim x As Integer = 1
Dim y As Integer = 1
Console.WriteLine(x.Equals(y))

y = x
Console.WriteLine(x.Equals(y))
```

Structures are value types, so they use bitwise comparison as well. Look at the following structure declaration.

**C#**

```
struct StructA
{
    public StructA(int a)
    {
        FieldA = a;
    }
    public int FieldA;
}
```

**Visual Basic**

```
Public Structure StructA
    Public Sub New(ByVal a As Integer)
        FieldA = a
    End Sub
    Public FieldA As Integer
End Structure
```

The following code snippet would display *false*.

**C#**

```
StructA x = new StructA(1);
StructA y = new StructA(2);

Console.WriteLine(x.Equals(y));
```

**Visual Basic**

```
Dim x As StructA = New StructA(1)
Dim y As StructA = New StructA(2)

Console.WriteLine(x.Equals(y))
```

The variables *x* and *y* are not binary-compatible because one contains a 1 and the other contains a 2. The following would be binary compatible, and would display *true*:

### C#

```
StructA x = new StructA(1);
StructA z = new StructA(1);
Console.WriteLine(x.Equals(z));
StructA y = x;
Console.WriteLine(x.Equals(y));
```

### Visual Basic

```
Dim x As StructA = New StructA(1)
Dim z As StructA = New StructA(1)

Console.WriteLine(x.Equals(z))
Dim y As StructA = x
Console.WriteLine(x.Equals(y))
```



**Note** An object is free to override the *Object.Equals(Object)* method. That is the case with the *String* class, which overrides the *Object.Equals(Object)* method so that it compares the internal strings themselves instead of the object references.

### Note to Implementers

The class or structure you are using for *T* should implement *IEquatable* (or preferably *IEquatable(T)*) interface and override the *Object.Equals(object)* method. Both should return the same result for the same object instance. Methods that need to compare instances of your value type need to box your value type to use the *Object.Equals(object)* method or *IEquatable* interface. They do not need to box if your implementation implements the *IEquatable(T)* interface and the code that does the comparison checks for the *IEquatable(T)* interface first.

## Understanding the Ordering Comparer

Some of the classes in the *System.Collections.Generic* namespace use what is called a *default ordering comparer*. This means that the method checks to see whether *T* implements the *IComparable(T)* interface. If so, the method calls the *CompareTo(T)* method on that interface. If not, the method checks to see whether the *IComparable* interface is defined, and if so, it then calls *CompareTo(object)* on the interface. When a class implements neither of these interfaces, the methods throw an error unless the caller specifies a comparison object to use.

### Note to Implementers

The class or structure you are using for  $T$  should implement the *IComparer* or preferably *IComparer*( $T$ ) interface—especially if you want your objects to be sorted or used in other comparison operations.

## Understanding Delegates, Anonymous Methods, and Lambda Expressions

Delegates were introduced in the Microsoft .NET Framework 1.0 as a type-safe method reference. They are used to reference a method that you want to pass to an event, a callback method, and so on. Anonymous methods were introduced in the .NET Framework 2.0 as a way of declaring such methods inline rather than placing the method code in a separate method. The following example defines the variable *Operation* using a named method.

### C#

```
class MethodDelegateClass
{
    int Square(int x) { return x * x; }

    public void Run()
    {
        Func<int, int> Operation = Square;

        Console.WriteLine("The square of 2 is {0}", Operation(2));
    }
}
```

### Visual Basic

```
Public Class MethodDelegateClass
    Function Square(ByVal x As Integer)
        Return x * x
    End Function

    Public Sub Run()
        Dim Operation As Func(Of Integer, Integer) = AddressOf Square

        Console.WriteLine("The square of 2 is {0}", Operation(2))
    End Sub
End Class
```

### Output

The square of 2 is 4

However, using the anonymous method feature, you can now define the method referenced by the variable *Operation* "anonymously" directly in your code block as follows.

#### C#

```
Func<int, int> Operation = delegate(int x) { return x * x; };

Console.WriteLine("The square of 2 is {0}", Operation(2));
```



**Note** Anonymous methods are not supported in Microsoft Visual Basic, but lambda expressions are, as you'll see in the section "Lambda Expressions in Visual Basic."

#### Output

The square of 2 is 4

Developers can use this to define simple methods in code without having to create a named method such as the following.

#### C#

```
System.Windows.Forms.Form form = (System.Windows.Forms.Form)obj;
form.Deactivate += delegate(object sender, EventArgs e)
{
    m_active.Remove(form);
};
```

In the .NET Framework 3.0, Microsoft added support for *lambda expressions* as an easier way to read and express an inline method. Lambda expressions are now preferred over anonymous methods as the best way to write an inline method. Here's an example of how to write a lambda expression.

#### C#

```
Func<int, int> Operation = x => x * x;

System.Console.WriteLine("The square of 2 is {0}", Operation(2));
```



**Note** See the following section for information about lambda expressions in Visual Basic.

#### Output

The square of 2 is 4

The preceding example uses a lambda expression to define a square function inline and assigns it to the variable *Operation*. *Operation* can then be called to square an integer.

## Lambda Expressions in Visual Basic

Lambda expressions were introduced to Visual Basic in Visual Basic 2008 (Visual Basic 9.0). In Visual Basic 2008, lambda expressions could only be one line and could only be functions. Visual Basic 2010 (Visual Basic 10) added support for multiple-line lambda expressions and allowed them to be either a sub or a function. The majority of lambda expressions written in this book compile only in Visual Basic 2010 but can be used with .NET Framework versions earlier than the .NET Framework 4. The following code demonstrates the lambda expression example shown earlier.

### Visual Basic 2008

```
Dim Operation as Func(Of Integer, Integer) = Function(x) x * x  
Console.WriteLine("The square of 2 is {0}", Operation(2))
```

However, you cannot write the following code in Visual Basic 2008 because it uses a sub.

### Visual Basic (Not in 2008)

```
Dim Operation As Action(Of Integer) = Sub(x) Console.WriteLine("The square of {0} is {1}", _  
x, x * x)
```

Only functions and one-line lambda expressions are allowed. With Visual Basic 2010, you can write lambda subs and functions as well as multiline lambda statements.

### Visual Basic 2010

```
Dim Operation As Action(Of Integer) = Sub(x) Console.WriteLine("The square of {0} is {1}", _  
x, x * x)
```

```
Operation(2)
```

Here's an example of a multiline statement.

### Visual Basic 2010

```
Dim Operation As Action(Of Integer) = Sub(x)  
    Dim squared As Integer = x * x  
    Console.WriteLine("The square of {0} is {1}",  
        x, squared)  
End Sub
```

```
Operation(2)
```

### Output

The square of 2 is 4



**Note** This book marks examples that can be compiled in only Visual Basic 2010 by using the header "Visual Basic 2010," but it uses only "Visual Basic" when you can compile the example in both Visual Basic 2008 and Visual Basic 2010.

## List(*T*) Overview

The *List(T)* class is a generic implementation of a dynamic array. (Dynamic arrays are discussed briefly in Chapter 1, “Understanding Collections: Arrays and Linked Lists.”) In addition to being a dynamic array, the class also lets you search, traverse, and sort elements. The *Contains*, *IndexOf*, *LastIndexOf*, and *Remove* methods use the default equality comparer, and the *BinarySearch* and *Sort* methods use an ordering comparer. Both comparers are discussed in the “Understanding the Equality and Ordering Comparers” section earlier in this chapter.

The *List(T)* class does not automatically sort. You must sort the list manually by calling the *Sort* method. If you want the list to sort the items for you automatically, you should look at the *SortedList(TKey,TValue)* and *SortedDictionary(TKey,TValue)* classes, which are discussed in Chapter 5, “Generic and Support Collections.”

Adding an item to a list that has *Count* equal to *Capacity* takes  $O(n)$  operations; otherwise, it takes only  $O(1)$  operations. Depending on the location where you’re inserting the item, inserting is similar to an *Add* and takes up to  $O(n)$  operations. Removing an item from the list takes up to  $O(n)$  operations.

The *List(T)* class is the generic version of the *ArrayList* class, so it does not have to box value types like the *ArrayList* class has to. According to MSDN, the amount of space saved by not boxing approximately 500 elements is greater than the memory used when generating a *List(T)* to handle the value type you specified for *T*.

## Using the List(*T*) Class

You can refer to the “Advantages of Arrays” and “Disadvantages of Arrays” sections in Chapter 1 to determine when to use the *List(T)* class, but in general, you would use *List(T)* when you have a fairly static list or a small number of elements. Because *List(T)* is the preferred way of creating a list, *ArrayList* is not discussed here. This book represents items in the list by using the notation  $[E_0, E_1, E_2, E_3 \dots E_N]$ , where  $E_0$  is element 0,  $E_1$  is element 1, and so on.

### Creating a List(*T*)

A *List(T)* can be created using one of three constructors: *List(T)()*, *List(T)(IEnumerable(T) collection)*, and *List(T)(int size)*.

#### *List(T)()*

You can create an empty *List(T)* instance by writing the following.

C#

```
List<int> a = new List<int>();
```

**Visual Basic**

```
Dim a As New List(Of Integer)
```

The preceding line creates an empty integer list, `[]`.

***List(T)(IEnumerable(T) collection)***

You can create a list with default values by passing an object that implements the `IEnumerable(T)` interface.

**C#**

```
List<int> a = new List<int>( new int [] { 2 , 4, 5 } );
```

**Visual Basic**

```
Dim a As New List(Of Integer)(New Integer() {2, 4, 5})
```

This code creates the list `[2,4,5]`.

***List(T)(int size)***

You can also create a list that has an initial internal array size, which is useful if you know in advance how many items you plan on adding to the list.

**C#**

```
List<int> a = new List<int>(100);
```

**Visual Basic**

```
Dim a As New List(Of Integer)(100)
```

This code creates a list of *ints* with an initial capacity of 100.

## Appending Items to a *List(T)*

To add (append) items to the end of a list, use the `Add` and `AddRange` methods.

***void Add(T item)***

The `Add` method takes the item that you want to add as the argument and adds it to end of the list.

**C#**

```
List<int> list = new List<int>();  
list.Add(2);  
list.Add(4);
```

**Visual Basic**

```
Dim list As New List(Of Integer)()

list.Add(2)
list.Add(4)
```

The list now contains the values [2,4].

***void AddRange(IEnumerable(T) collection)***

In addition to single items, you can add a collection of items by using the *AddRange* method. The *AddRange* method takes the collection that you want to add as the argument. The *collection* must implement the *IEnumerable(T)* interface. The majority of the classes in *System.Collections.Generic* implement *IEnumerable(T)* along with the *T []* array.

**C#**

```
List<int> list = new List<int>();
list.AddRange( new int []{ 2, 4 } );
```

**Visual Basic**

```
Dim list As New List(Of Integer)()

list.AddRange(New Integer() {2, 4})
```

The list now contains the values [2,4].

## Traversing a *List(T)*

You traverse a *List(T)* by either calling the *For Each* method (Visual Basic) or the *foreach* statement (C#).

### The *foreach* Statement

The *List(T)* class inherits the *IEnumerable* interface, which means you can use the class with the *foreach* statement.



**More Info** Chapter 6, “.NET Collection Interfaces,” provides more information about the *IEnumerable* interface. Also, in Chapter 7, “Introduction to LINQ,” you will learn about LINQ, a query mechanism that you can use to make SQL-like queries on collections.

The following code shows an example of using the *foreach* statement with a list.

#### C#

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 4, 5, 6, 7 });
foreach (int value in list)
{
    Console.WriteLine(value);
}
```

#### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7})

For Each value As Integer In list
    Console.WriteLine(value)
Next
```

The code writes each element of the list on a separate line, as follows.

#### Output

```
1
2
3
4
5
6
7
```

### **void ForEach(Action(T) action)**

The *ForEach* method performs the action passed as *Action(T)* on each element in the list. *Action(T)* is a method defined as follows.

#### C#

```
public delegate void Action<T>(T obj);
```

#### Visual Basic

```
Public Delegate Sub Action(Of T)(ByVal obj As T)
```

**Using a Method for the *Action(T)* Argument** Suppose you have a group of tasks that need to be processed by an external method called *ProcessTask*. You can pass the *ProcessTask* method as an *Action(T)* using the following code.

Your task object is defined as the following.

C#

```
class Task
{
    public Task(string name)
    {
        Name = name;
    }
    public string Name;
    public bool IsProcessed;
}
```

Visual Basic

```
Public Class Task
    Public Sub New(ByVal name As String)
        Me.Name = name
    End Sub
    Public Name As String
    Public IsProcessed As Boolean
End Class
```

Your method declaration of *ProcessTask* is as follows.

C#

```
static void ProcessTask(Task task)
{
    // Nothing to process if the task is null or has already been processed
    if (task == null || task.IsProcessed)
    {
        return;
    }

    // Process the task
    // ...

    // Mark the task as processed
    task.IsProcessed = true;
}
```

Visual Basic

```
Public Sub ProcessTask(ByVal task As Task)
    ' Nothing to process if the task is null or has already been processed
    If (task Is Nothing) Then
        Exit Sub
    End If

    If (task.IsProcessed) Then
        Exit Sub
    End If
```

```
' Process the task
' ...

' Mark the task as processed
task.IsProcessed = True
End Sub
```

Now you can use the following code to process the task.

### C#

```
List<Task> list = new List<Task>();
list.Add(new Task("BACKUP"));
list.Add(new Task("VERIFY"));
list.Add(new Task("MARK_ARCHIVED"));
list.ForEach(ProcessTask);
```

### Visual Basic

```
Dim list As New List(Of Task)
list.Add(New Task("BACKUP"))
list.Add(New Task("VERIFY"))
list.Add(New Task("MARK_ARCHIVED"))
list.ForEach(AddressOf ProcessTask)
```

The code calls the *ProcessTask* method on each element in the list and marks its *IsProcessed* field as *true*.

**Using a Lambda Expression for the *Action(T)* Method** For simple operations, it's not always convenient to go through the trouble of creating a separate method. Fortunately, you can use a lambda expression to perform the action instead. The following code uses a lambda expression to sum all the elements in a list.

### C#

```
List<int> list = new List<int>(new int[] { 16, 8, 44, 5 });
int sum = 0;
list.ForEach(value =>
{
    sum += value;
});
Console.WriteLine(sum);
```

### Visual Basic 2010

```
Dim aList As New List(Of Integer)(New Integer() {16, 8, 44, 5})
Dim sum As Integer

aList.ForEach(Sub(value As Integer)
    sum = sum + value
End Sub)
Console.WriteLine(sum)
```

### Output

73

For each element, the method calls the lambda expression, which adds that element's value to the *sum* variable. In this case, the *sum* variable contains the value 73 after the *ForEach* method completes—the sum of all elements in *list*. You could, of course, write a lambda expression to do other things, such as count all the even numbers.

## Removing Items from a *List(T)*

You can remove items from a list by using the *Clear*, *Remove*, *RemoveAll*, *RemoveAt*, and *RemoveRange* methods. None of these methods modify the capacity of the list. To modify the capacity of the list, you need to call the *TrimExcess* method, discussed later in this chapter.

### **void Clear()**

The *Clear* method removes all items from the list.

#### C#

```
List<int> list = new List<int>(new int[] { 1, 6, 2, 3 });
list.Clear();
```

#### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {1, 6, 2, 3})
list.Clear()
```

The list will be empty, `[]`, after the *Clear* method is called. (But just as a reminder, the list will still have the same capacity.)

### **bool Remove(*T item*)**

The *Remove* method removes a specified item from the list. The method returns *true* when the item is removed successfully, or *false* if the specified item either isn't found or isn't removed. Internally, the list locates items using the equality comparer discussed in the "Understanding the Equality and Ordering Comparers" section in this chapter.

#### C#

```
List<int> list = new List<int>(new int[] { 1, 6, 2, 3 });
list.Remove(6);
```

#### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {1, 6, 2, 3})
list.Remove(6)
```

The preceding code changes the list from `[ 1, 6, 2, 3 ]` to `[ 1, 2, 3 ]` because a 6 was passed in to the *Remove* method.

***int RemoveAll(Predicate(T) match)***

*RemoveAll* removes all items that match the specified predicate. The *Predicate(T)* is defined as follows.

**C#**

```
delegate bool Predicate<T>(T obj);
```

**Visual Basic**

```
Dim instance As New Predicate(Of T)(AddressOf HandlerMethod) as Boolean
```

*Predicate(T)* returns *true* when an object matches the criteria, meaning that the object should be removed. The *RemoveAll* method returns the number of items removed.

**Using a Method with *RemoveAll*** The following code removes all even values by using a method for the *Predicate(T)*.

**C#**

```
static bool IsEven(int value)
{
    return (value % 2) == 0;
}
```

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
list.RemoveAll(IsEven);
```

**Visual Basic**

```
Function IsEven(ByVal value As Integer) As Boolean
    Return ((value Mod 2) = 0)
End Function
```

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})
list.RemoveAll(AddressOf IsEven)
```

The list contains [ 1, 3, 5, 7, 9 ] after the *RemoveAll* is called.

**Using a Lambda Expression with *RemoveAll*** The following code removes all even values by using a lambda expression for the *Predicate(T)*.

**C#**

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
list.RemoveAll(item => { return (item % 2) == 0; });
```

**Visual Basic**

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})
list.RemoveAll(Function(x) x Mod 2 = 0)
```

The list contains [ 1, 3, 5, 7, 9 ] after the *RemoveAll* method completes.

### **void RemoveAt(int index)**

The *RemoveAt* method removes the item at a specified index. The method throws an exception if the index passed in is out of range.

#### C#

```
List<string> list = new List<string>(new string[] { "red", "blue", "green" });
list.RemoveAt(0);
```

#### Visual Basic

```
Dim list As New List(Of String)(New String() {"red", "blue", "green"})
list.RemoveAt(0)
```

The list contains only [ "blue", "green" ] after calling *RemoveAt* in the preceding code, because the index that was passed in, 0, is the position of element "red".

### **void RemoveRange(int index, int count)**

The *RemoveRange* method removes a specified range of elements from the list. The elements that will be removed are the elements that are  $\geq index$  and  $\leq (index + count - 1)$ .

#### C#

```
List<string> list = new List<string>(new string[] { "red", "please", "remove", "me", "blue",
"green" });
list.RemoveRange(1,3);
```

#### Visual Basic

```
Dim list As New List(Of String) _
(New String() {"red", "please", "remove", "me", "blue", "green"})
list.RemoveRange(1, 3)
```

The list contains only [ "red", "blue", "green" ] after calling *RemoveRange* in the preceding code, because "please", "remove", and "me" occupied indices 1, 2, and 3 respectively.

## **Inserting Items into a *List(T)***

You can insert items into the list by calling the *Insert* and *InsertRange* methods. The item at that specified index location and all items with a higher index shift to the right to accommodate the new item.

### **void Insert(int index, T item)**

The *Insert* method inserts an item into the list at the specified location. Note that specifying an index equal to *Count* is the same as calling the *Add* method.

#### C#

```
List<string> list = new List<string>(new string[] { "Adam", "Charlie", "David" });
list.Insert(1, "Bob");
```

#### Visual Basic

```
Dim list As New List(Of String)(New String() {"Adam", "Charlie", "David"})
list.Insert(1, "Bob")
```

After running the preceding code, the list contains *["Adam", "Bob", "Charlie", "David"]*, because the code inserts *"Bob"* at the element 1 position, which formerly contained the value *"Charlie"*.

### **void InsertRange(int index, IEnumerable collection)**

The *InsertRange* method inserts a collection of items into the list at a specified location. Specifying an index equal to *Count* is the same as calling the *AddRange* method. The method adds items to the collection in the order they are enumerated.

#### C#

```
List<string> list = new List<string>(new string[] { "Adam", "Charlie", "David" });
List<string> listToAppend = new List<string>(new string[] { "Bob", "Bobbie", "Bobby" });
list.InsertRange(1, listToAppend);
```

#### Visual Basic

```
Dim list As List(Of String) = New List(Of String)(New String() {"Adam", "Charlie", "David"})
Dim listToAppend As New List(Of String)(New String() {"Bob", "Bobbie", "Bobby"})
list.InsertRange(1, listToAppend)
```

The list contains *["Adam", "Bob", "Bobbie", "Bobby", "Charlie", "David"]* after the *InsertRange* call in the preceding code completes, because the list *["Bob", "Bobbie", "Bobby"]* was inserted at element 1, which formerly contained the value *"Charlie"*.

## **Sorting a List(*T*)**

You can sort a list by calling one of the four overloaded *Sort* methods. In fact, you must call one of the *Sort* methods manually whenever you want the list to be sorted, because *List(*T*)* doesn't support automatic sorting.

## How Built-in Sorting Works

The *List(T)* class performs sorts by using the Quicksort algorithm from *Array.Sort*. The Quicksort algorithm picks an element in the list that will be a *pivot* point. All items less than the pivot point are placed before the pivot point, and all items greater than the pivot point are placed after it. Any value equal to the pivot point can go before or after the pivot point. The algorithm then repeats, sorting the left and right sides recursively the same way. Here's a simple example of the algorithm.

Before starting, the *QuickSort* example uses this *ListToString* method, which writes the contents of a list to a string. This is handy for showing the contents of a list on the screen.

### C#

```
static StringBuilder ListToString(System.Collections.IList array)
{
    StringBuilder sb = new StringBuilder();
    sb.Append("[");
    for (int i = 0; i < array.Count - 1; ++i)
    {
        sb.Append(array[i]);
        sb.Append(",");
    }
    sb.Append(array[array.Count - 1]);
    sb.Append("]");
}

return sb;
}
```

### Visual Basic

```
Function ListToString(ByVal array As System.Collections.IList) As System.Text.StringBuilder
    Dim sb As New System.Text.StringBuilder()
    sb.Append("[")
    For i As Integer = 0 To array.Count - 2
        sb.Append(array(i))
        sb.Append(",")
    Next
    sb.Append(array(array.Count - 1))
    sb.Append("]")

    Return sb
End Function
```

This function writes an object that implements *IList* to a *StringBuilder* in the format “[ $E_0 E_1 E_2 E_3 \dots E_N$ ]”, where  $E_0$  is element 0,  $E_1$  is element 1, and so on.

Here's the *QuickSort* method example.

C#

```
static void QuickSort(List<int> items)
{
    // There isn't a need to sort an array that contains one element.
    if (items.Count <= 1)
    {
        return;
    }

    // Create a list that holds the values that are less than
    // and greater than the pivot point
    List<int> less = new List<int>();
    List<int> greater = new List<int>();

    // Use the last point as the pivot point
    int pivot = items.Last();

    // Traverse the list and set the pivot point to the lesser than or greater than list
    // There isn't a need to check the last one since it is the pivot point
    for (int i = 0; i < items.Count - 1; ++i)
    {
        if (items[i] < pivot)
        {
            less.Add(items[i]);
        }
        else
        {
            greater.Add(items[i]);
        }
    }

    // Sort the new list
    QuickSort(less);
    QuickSort(greater);

    int index = 0;

    // Add the items that are less than the pivot point to the beginning
    foreach (int value in less)
    {
        items[index++] = value;
    }

    // Add the pivot point
    items[index++] = pivot;

    // Add the items that are greater than the pivot point to the end
    foreach (int value in greater)
    {
        items[index++] = value;
    }
}
```

**Visual Basic**

```
Sub QuickSort(ByVal items As List(Of Integer))
    ' There isn't a need to sort an array that contains one element.
    If (items.Count <= 1) Then
        Return
    End If

    ' Create a list that holds the values that are less than
    ' and greater than the pivot point
    Dim less As New List(Of Integer)
    Dim greater As New List(Of Integer)

    ' Use the last point as the pivot point
    Dim pivot As Integer = items.Last()

    ' Traverse the list and set the pivot point to the lesser than or greater than list
    ' There isn't a need to check the last one since it is the pivot point
    For i As Integer = 0 To items.Count - 2
        If (items(i) < pivot) Then
            less.Add(items(i))
        Else
            greater.Add(items(i))
        End If
    Next

    ' Sort the new list
    QuickSort(less)
    QuickSort(greater)

    Dim index As Integer = 0

    ' Add the items that are less than the pivot point to the beginning
    For Each value As Integer In less
        items(index) = value
        index = index + 1
    Next

    ' Add the pivot point
    items(index) = pivot
    index = index + 1

    ' Add the items that are greater than the pivot point to the end
    For Each value As Integer In greater
        items(index) = value
        index = index + 1
    Next
End Sub
```

The *QuickSort* method obtains a pivot point by picking the last item in the list to be sorted. Next, it traverses the list and adds the element's value to either the *greater* or *less* list by comparing the value to the pivot point. The method then recursively calls the *QuickSort* method on each list, and finally, combines the list.



**Note** This example function is designed to show you how *QuickSort* works; it isn't optimized or designed to win any coder-of-the-year awards.

The following code drives the sort operation.

#### C#

```
// Create a random list of 20 integers
Random rnd = new Random();
List<int> items = Enumerable.Repeat(0, 20).Select(i => rnd.Next(100)).ToList();

// Write the new list to the screen
Console.WriteLine("items={0}",ListToString(items));

// Sort the list
QuickSort(items);

// Write the sorted list to the screen
Console.WriteLine("QuickSort(items)={0}", ListToString(items));
```

#### Visual Basic

```
' Create a random list of 20 integers
Dim rnd As Random = New Random()
Dim items As Enumerable.Repeat(0, 20).Select(Function(i) rnd.Next(100)).ToList()

' Write the new list to the screen
Console.WriteLine("items={0}", ListToString(items))

' Sort the list
QuickSort(items)

' Write the sorted list to the screen
Console.WriteLine("QuickSort(items)={0}", ListToString(items))
```

#### Output

```
items=[36,96,35,60,85,53,29,5,62,22,84,80,23,91,22,5,90,17,59,24]

QuickSort(items)=[5,5,17,22,22,23,24,29,35,36,53,59,60,62,80,84,85,90,91,96]
```

The first line of the output is the unsorted list of random numbers. The second line is the list after it has been quick-sorted.

### void Sort()

The *Sort* method uses the default comparer as described in the "Understanding the Equality and Ordering Comparers" section earlier in this chapter.

#### C#

```
List<int> list = new List<int>(new int[] { 16, 8, 44, 5 });
list.Sort();
```

#### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {16, 8, 44, 5})
list.Sort()
```

The *Sort* method sorts the list to [5, 8, 16, 44].

### void Sort(Comparison(T) comparison)

The *Sort(Comparison(T))* method uses the *Comparison(T)* method to do the sort. The *Comparison(T)* method is defined as *int Comparison(T x, T y)*. The method must return -1 when *x* is less than *y*, 1 when *x* is greater than *y*, and 0 when *x* equals *y*.

The following examples both use the *Person* struct.

#### C#

```
struct Person
{
    /// <summary>
    /// The age of the person.
    /// </summary>
    public int Age;
    /// <summary>
    /// The first name of the person.
    /// </summary>
    public string FirstName;
    /// <summary>
    /// The last name of the person.
    /// </summary>
    public string LastName;
    /// <summary>
    /// Initializes a new instance of the Person structure with the specified contents.
    /// </summary>
    /// <param name="firstName">The first name of the person.</param>
    /// <param name="lastName">The last name of the person.</param>
    /// <param name="age">The age of the person.</param>
    public Person(string firstName, string lastName, int age)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
}
```

```
/// <summary>
/// Override the ToString method for display purposes.
/// </summary>
/// <returns></returns>
public override string ToString()
{
    return string.Format("[{0},{1},{2}]", FirstName, LastName, Age);
}
```

## Visual Basic

```
Structure Person
    ''' <summary>
    ''' The age of the person.
    ''' </summary>
    Public Age As Integer
    ''' <summary>
    ''' The first name of the person.
    ''' </summary>
    Public FirstName As String
    ''' <summary>
    ''' The last name of the person.
    ''' </summary>
    Public LastName As String
    ''' <summary>
    ''' Initializes a new instance of the Person structure with the specified contents.
    ''' </summary>
    ''' <param name="firstName">The first name of the person.</param>
    ''' <param name="lastName">The last name of the person.</param>
    ''' <param name="age">The age of the person.</param>
    Public Sub New(ByVal firstName As String, ByVal lastName As String, _
                  ByVal age As Integer)
        Me.firstName = firstName
        Me.lastName = lastName
        Me.Age = age
    End Sub
    ''' <summary>
    ''' Override the ToString method for display purposes.
    ''' </summary>
    ''' <returns></returns>
    Public Overrides Function ToString() As String
        Return String.Format("[{0},{1},{2}]", FirstName, LastName, Age)
    End Function
End Structure
```



**Note** The example structure could just as easily be a class. Both work the same way for sorting purposes.

**Using a Lambda Expression for the *Comparison(T)*** The following is an example that uses a lambda expression to do the comparison. The lambda expression compares the ages of each person and sorts the list accordingly.

#### C#

```
List<Person> list = new List<Person>();
list.Add(new Person("David", "Alexander", 44));
list.Add(new Person("Jeff", "Hay", 32));
list.Add(new Person("Kim", "Akers", 12));
list.Add(new Person("Michael", "Allen", 25));
list.Sort((a, b) =>
{
    if (a.Age < b.Age)
    {
        return -1;
    }

    if (a.Age > b.Age)
    {
        return 1;
    }

    return 0;
});

```

#### Visual Basic 2010

```
Dim list As New List(Of Person)()

List.Add(New Person("David", "Alexander", 44))
List.Add(New Person("Jeff", "Hay", 32))
List.Add(New Person("Kim", "Akers", 12))
List.Add(New Person("Michael", "Allen", 25))

list.Sort(Function(a, b)
    If (a.Age < b.Age) Then
        Return -1
    End If

    If (a.Age > b.Age) Then
        Return 1
    End If
    Return 0
End Function
)
```

The list contains `[[Kim,Akers,12],[Michael,Allen,25],[Jeff,Hay,32],[David,Alexander,44]]` after the example executes.

**Using a Method for the *Comparison(T)*** Here's an example that uses a method to do the comparison.

**C#**

```
static int AgeComparison(Person a, Person b)
{
    if (a.Age < b.Age)
    {
        return -1;
    }

    if (a.Age > b.Age)
    {
        return 1;
    }

    return 0;
}
```

**Visual Basic**

```
Function AgeComparison(ByVal a As Person, ByVal b As Person) As Integer
    If (a.Age < b.Age) Then
        Return -1
    End If

    If (a.Age > b.Age) Then
        Return 1
    End If

    Return 0
End Function
```

The *AgeComparison* method compares the ages of the two *Person* structures passed in. The following code uses the *AgeComparison* method to sort the list.

**C#**

```
List<Person> list = new List<Person>();
list.Add(new Person("David", "Alexander", 44));
list.Add(new Person("Jeff", "Hay", 32));
list.Add(new Person("Kim", "Akers", 12));
list.Add(new Person("Michael", "Allen", 25));
list.Sort(AgeComparison);
```

**Visual Basic**

```
Dim list As List(Of Person) = New List(Of Person)()

list.Add(New Person("David", "Alexander", 44))
list.Add(New Person("Jeff", "Hay", 32))
list.Add(New Person("Kim", "Akers", 12))
list.Add(New Person("Michael", "Allen", 25))

list.Sort(AddressOf AgeComparison)
```

When executed, the list will contain *[[Kim,Akers,12],[Michael,Allen,25],[Jeff,Hay,32],[David,Alexander,44]]*.

The method *AgeComparison* compares the age of each person and sorts the list accordingly.

### **void Sort(IComparer(T) comparer)**

The *Sort(IComparer(T))* method uses the interface *IComparer(T)* for the comparison.

The following example uses the *Person* structure defined in the *Sort(Comparison(T))* method. To use the *Sort(IComparer(T))* method, you must first define a data type that implements the *IComparer(T)* interface, as follows.

#### C#

```
// <summary>
/// Defines a class for doing age comparisons on the person structure.
/// </summary>
class PersonAgeComparer : IComparer<Person>
{
    /// <summary>
    /// Compares two people using their ages.
    /// </summary>
    /// <param name="a">The first person to compare.</param>
    /// <param name="b">The second person to compare.</param>
    /// <returns>Returns a -1 if the first person's age is less than the second person,
    /// 0 if they are the same,
    /// and 1 if the first person is greater than the second.</returns>
    public int Compare(Person a, Person b)
    {
        if (a.Age < b.Age)
        {
            return -1;
        }

        if (a.Age > b.Age)
        {
            return 1;
        }

        return 0;
    }
}
```

#### Visual Basic

```
''' <summary>
''' Defines a class for doing age comparisons on the person structure.
''' </summary>
Class PersonAgeComparer
    Implements IComparer(Of Person)
```

```
''' <summary>
''' Compares two people using their ages.
''' </summary>
''' <param name="a">The first person to compare.</param>
''' <param name="b">The second person to compare.</param>
''' <returns>Returns a -1 if the first person's age is less than the second person,
''' 0 if they are the same,
''' and 1 if the first person is greater than the second.</returns>
Public Function Compare(ByVal a As Person, ByVal b As Person) As Integer _
    Implements IComparer(Of Person).Compare
    If (a.Age < b.Age) Then
        Return -1
    End If

    If (a.Age > b.Age) Then
        Return 1
    End If

    Return 0
End Function
End Class
```

The following code uses the *PersonAgeComparer* class to do the comparison. The class implements the interface *IComparer(T)*, which contains the method *Compare*. The *IComparer(T)*.*Compare* method is called by the *Sort* method. The *PersonAgeComparer* class can be used by the following driver.

### C#

```
PersonAgeComparer comparer = new PersonAgeComparer();

List<Person> list = new List<Person>();
list.Add(new Person("David", "Alexander", 44));
list.Add(new Person("Jeff", "Hay", 32));
list.Add(new Person("Kim", "Akers", 12));
list.Add(new Person("Michael", "Allen", 25));
list.Sort(comparer);
```

### Visual Basic

```
Dim comparer As PersonAgeComparer = New PersonAgeComparer()

Dim list As List(Of Person) = New List(Of Person)()
list.Add(New Person("David", "Alexander", 44))
list.Add(New Person("Jeff", "Hay", 32))
list.Add(New Person("Kim", "Akers", 12))
list.Add(New Person("Michael", "Allen", 25))
list.Sort(comparer)
```

After this code executes, the list contains `[[Kim,Akers,12],[Michael,Allen,25],[Jeff,Hay,32],[David,Alexander,44]]`. The `PersonAgeComparer` class compares the ages of each person and sorts the list accordingly.

### **void Sort(int index, int count, IComparer(T) comparer)**

With the `Sort(int, int, IComparer(T))` method, you can specify a range of elements to be sorted in the list.

To do this, you need to extend the `Person` structure defined earlier to include the sex of the person by adding the following.

#### C#

```
/// <summary>
/// States if the person is male.
/// </summary>
public bool IsMale;
```

#### Visual Basic

```
'<summary>
'<summary> States if the person is male.
'</summary>
public IsMale as Boolean
```

Then change the constructor and `toString` from the following:

#### C#

```
/// <summary>
/// Initializes a new instance of the Person structure with the specified contents.
/// </summary>
/// <param name="firstName">The first name of the person.</param>
/// <param name="lastName">The last name of the person.</param>
/// <param name="age">The age of the person.</param>
public Person(string firstName, string lastName, int age)
{
    FirstName = firstName;
    LastName = lastName;
    Age = age;
}
/// <summary>
/// Override the toString method for display purposes.
/// </summary>
/// <returns></returns>
public override string ToString()
{
    return string.Format("[{0},{1},{2}]",FirstName,LastName,Age);
}
```

### Visual Basic

```
''' <summary>
''' Initializes a new instance of the Person structure with the specified contents.
''' </summary>
''' <param name="firstName">The first name of the person.</param>
''' <param name="lastName">The last name of the person.</param>
''' <param name="age">The age of the person.</param>
Public Sub New(ByVal firstName As String, ByVal lastName As String, ByVal age As Integer)
    Me.firstName = firstName
    Me.lastName = lastName
    Me.Age = age
End Sub
''' <summary>
''' Override the ToString method for display purposes.
''' </summary>
''' <returns></returns>
Public Overrides Function ToString() As String
    Return String.Format("[{0},{1},{2}]", FirstName, LastName, Age)
End Function
```

to this:

### C#

```
/// <summary>
/// Initializes a new instance of the Person structure with the specified contents.
/// </summary>
/// <param name="firstName">The first name of the person.</param>
/// <param name="lastName">The last name of the person.</param>
/// <param name="age">The age of the person.</param>
/// <param name="isMale">States if the person is male.</param>
public Person(string firstName, string lastName, int age, bool isMale)
{
    FirstName = firstName;
    LastName = lastName;
    Age = age;
    IsMale = isMale;
}
/// <summary>
/// Override the ToString method for display purposes.
/// </summary>
/// <returns></returns>
public override string ToString()
{
    return string.Format("[{0},{1},{2},{3}]", FirstName, LastName, Age, IsMale ? "M" : "F");
}
```

### Visual Basic

```
''' <summary>
''' Initializes a new instance of the Person structure with the specified contents.
''' </summary>
''' <param name="firstName">The first name of the person.</param>
''' <param name="lastName">The last name of the person.</param>
''' <param name="age">The age of the person.</param>
''' <param name="isMale">States if the person is male.</param>
```

```

Public Sub New(ByVal firstName As String, ByVal lastName As String, ByVal age As Integer, _
    ByVal isMale As Boolean)
    Me.FirstName = firstName
    Me.LastName = lastName
    Me.Age = age
    Me.IsMale = isMale
End Sub
''' <summary>
''' Override the ToString method for display purposes.
''' </summary>
''' <returns></returns>
Public Overrides Function ToString() As String
    Return String.Format("[{0},{1},{2},{3}]", FirstName, LastName, Age, _
        If(IsMale, "M", "F"))
End Function

```

You also need to create a comparison class. For this example, the comparison class must compare names, so it's named *PersonNameComparer*.

### C#

```

// <summary>
/// Defines a class for doing name comparisons on the person structure
/// using the last name first.
/// </summary>
class PersonNameComparer : IComparer<Person>
{
    /// <summary>
    /// Compares two people using their names.
    /// </summary>
    /// <param name="a">The first person to compare.</param>
    /// <param name="b">The second person to compare.</param>
    /// <returns>Returns a -1 if the first person's name is less than the second person,
    /// 0 if they are the same,
    /// and 1 if the first person is greater than the second.</returns>
    public int Compare(Person a, Person b)
    {
        StringComparer comparer = StringComparer.CurrentCultureIgnoreCase;

        int lastNameComparison = comparer.Compare(a.LastName, b.LastName);

        // If the last names are not equal, there is no need to compare the first name.
        if (lastNameComparison != 0)
        {
            return lastNameComparison;
        }

        // Return the result of the first name comparison since the last name is equal
        return comparer.Compare(a.FirstName, b.FirstName);
    }
}

```

### Visual Basic

```

''' <summary>
''' Defines a class for doing name comparisons on the person structure
''' using the last name first.
''' </summary>
Class PersonNameComparer
    Implements IComparer(Of Person)
    ''' <summary>
    ''' Compares two people using their names.
    ''' </summary>
    ''' <param name="a">The first person to compare.</param>
    ''' <param name="b">The second person to compare.</param>
    ''' <returns>Returns a -1 if the first person's name is less than the second person,
    ''' 0 if they are the same,
    ''' and 1 if the first person is greater than the second.</returns>
    Public Function Compare(ByVal a As Person, ByVal b As Person) _
        As Integer Implements IComparer(Of Person).Compare
        Dim comparer As StringComparer = StringComparer.CurrentCultureIgnoreCase

        Dim lastNameComparison As Integer = comparer.Compare(a.LastName, b.LastName)

        ' If the last names are not equal, there is no need to compare the first name.
        If (lastNameComparison <> 0) Then
            Return lastNameComparison
        End If

        ' Return the result of the first name comparison since the last name is equal
        Return comparer.Compare(a.FirstName, b.FirstName)
    End Function
End Class

```

The *Compare* method returns a compare result by checking the last name and then the first name.

The driver creates an unsorted list of people and then sorts them by sex. Next, it sorts each sex group by name, using the *Sort(int, int, IComparer(T))* method and *PersonNameComparer*.

### C#

```

List<Person> list = new List<Person>();
list.Add(new Person("Annie", "Herriman", 12, false));
list.Add(new Person("David", "Alexander", 44, true));
list.Add(new Person("Michael", "Allen", 25, true));
list.Add(new Person("Lisa", "Andrews", 42, false));
list.Add(new Person("Jeff", "Hay", 32, true));
list.Add(new Person("Kim", "Akers", 28, false));

// First, sort everyone according to their sex
// The females will be at the front of the list
list.Sort((a, b) =>
{
    if (a.IsMale == b.IsMale)
    {
        return 0;
    }
}

```

```

        if (a.IsMale)
    {
        // returning a 1 will put the males after the females
        return 1;
    }
    return -1;
});

// You know the first three are females because we only have three females,
// so you can sort the females by calling the method with a count of 0
list.Sort(0, 3, new PersonNameComparer());
// Next sort the remaining 3 males
list.Sort(3, 3, new PersonNameComparer());

```

**Visual Basic 2010**

```

Dim list As List(Of Person) = New List(Of Person)()

list.Add(New Person("Annie", "Herriman", 12, False))
list.Add(New Person("David", "Alexander", 44, True))
list.Add(New Person("Michael", "Allen", 25, True))
list.Add(New Person("Lisa", "Andrews", 42, False))
list.Add(New Person("Jeff", "Hay", 32, True))
list.Add(New Person("Kim", "Akers", 28, False))

' First, sort everyone according to their sex
' The females will be at the front of the list

list.Sort(Function(a, b)
    If (a.IsMale = b.IsMale) Then
        Return 0
    End If
    If (a.IsMale) Then
        ' returning a 1 will put the males after the females
        Return 1
    End If
    Return -1
End Function
)

Console.WriteLine(ListToString(list))

' You know the first 3 are females because we only have 3 females,
' so you can sort the females by calling the method with a count of 0
list.Sort(0, 3, New PersonNameComparer())
' Next sort the remaining 3 males
list.Sort(3, 3, New PersonNameComparer())

```

After the code executes, the list contains `[[Kim,Akers,28,F],[Lisa,Andrews,42,F],[Annie,Herriman,12,F],[David,Alexander,44,M],[Michael,Allen,25,M],[Jeff,Hay,32,M]]`. After the first sort, the first three are female and the last three are male. The code then passes the female and then the male ranges to the `Sort(int, int, IComparer(T))` method to sort by name.

## Searching a *List(T)*

Use the *Find*, *FindIndex*, *FindAll*, and *BinarySearch* methods to find an item in the list.

### *T Find(Predicate(T) match) and T FindLast(Predicate(T) match)*

The *Find* and *FindLast* methods search the list for an item that matches the condition specified in the *match* argument. The *Find* method returns the first found occurrence, and *FindLast* returns the last found occurrence. Both methods return the default value when no match is found. The *Predicate(T)* method is defined as a method that takes an argument of type *T* and returns a *bool*. The method returns *true* when the argument matches the criteria, and *false* otherwise.

This example uses the following structure.

#### C#

```
struct Person
{
    ///<summary>
    ///<summary>
    ///</summary>
    public int Age;
    ///<summary>
    ///<summary>
    public string FirstName;
    ///<summary>
    ///<summary>
    public string LastName;
    ///<summary>
    ///<summary>
    public bool IsMale;
    ///<summary>
    ///<summary>
    public bool IsEmpty
    {
        get
        {
            return
                (string.IsNullOrEmpty(LastName) ||
                 string.IsNullOrEmpty(FirstName) ||
                 Age == 0);
        }
    }
    ///<summary>
    ///<summary>
    ///<param name="firstName">The first name of the person.</param>
    ///<param name="lastName">The last name of the person.</param>
    ///<param name="age">The age of the person.</param>
    ///<param name="isMale">States if the person is male.</param>
```

```

public Person(string firstName, string lastName, int age, bool isMale)
{
    FirstName = firstName;
    LastName = lastName;
    Age = age;
    IsMale = isMale;
}
/// <summary>
/// Override the ToString method for display purposes.
/// </summary>
/// <returns></returns>
public override string ToString()
{
    return string.Format("[{0},{1},{2}]", FirstName, LastName, Age, IsMale ? "M" : "F");
}
}

```

### Visual Basic

```

Structure Person
    ''' <summary>
    ''' The age of the person.
    ''' </summary>
    Public Age As Integer
    ''' <summary>
    ''' The first name of the person.
    ''' </summary>
    Public FirstName As String
    ''' <summary>
    ''' The last name of the person.
    ''' </summary>
    Public LastName As String
    ''' <summary>
    ''' States if the person is male.
    ''' </summary>
    Public IsMale As Boolean
    ''' <summary>
    ''' States if the data is empty.
    ''' </summary>
    Public ReadOnly Property IsEmpty As Boolean
        Get
            Return (String.IsNullOrEmpty(LastName) Or String.IsNullOrEmpty(FirstName) _
                    Or Age = 0)
        End Get
    End Property
    ''' <summary>
    ''' Initializes a new instance of the Person structure with the specified contents.
    ''' </summary>
    ''' <param name="firstName">The first name of the person.</param>
    ''' <param name="lastName">The last name of the person.</param>
    ''' <param name="age">The age of the person.</param>
    ''' <param name="isMale">States if the person is male.</param>
    Public Sub New(ByVal firstName As String, ByVal lastName As String, _
                  ByVal age As Integer, ByVal isMale As Boolean)

```

```
    Me.FirstName = firstName
    Me.LastName = lastName
    Me.Age = age
    Me.IsMale = isMale
End Sub
''' <summary>
''' Override the ToString method for display purposes.
''' </summary>
''' <returns></returns>
Public Overrides Function ToString() As String
    Return String.Format("[{0},{1},{2},{3}]", FirstName, LastName, Age, _
        If(IsMale, "M", "F"))
End Function
End Structure
```

**Using a Lambda Expression for the Search** You can use lambda expressions for the *Find* method. Say you want to find an object in a collection by using a complex criterion. Take a look at the following example.

### C#

```
List<Person> list = new List<Person>();
list.Add(new Person("Julia", "Llyina", 12, false));
list.Add(new Person("David", "Alexander", 44, true));
list.Add(new Person("Michael", "Allen", 25, true));
list.Add(new Person("Andrews", "Lisa", 42, false));
list.Add(new Person("Jeff", "Hay", 32, true));
list.Add(new Person("Kim", "Akers", 28, false));

Person found = list.Find(person =>
{
    return person.IsMale && person.Age > 40;
});
```

### Visual Basic

```
Dim comparer As PersonAgeComparer = New PersonAgeComparer()

Dim list As List(Of Person) = New List(Of Person)()
list.Add(New Person("Julia", "Llyina", 12, False))
list.Add(New Person("David", "Alexander", 44, True))
list.Add(New Person("Michael", "Allen", 25, True))
list.Add(New Person("Andrews", "Lisa", 42, False))
list.Add(New Person("Jeff", "Hay", 32, True))
list.Add(New Person("Kim", "Akers", 28, False))

Dim found As Person = list.Find(Function(person) person.IsMale And person.Age > 40)
```

This example finds a person in the list who is male and is over the age of 40. When executed, *["David","Alexander",44,M]* is assigned to the *found* variable. When the item is not found, as in the following search, the *found* variable gets assigned a default value.

**C#**

```
Person found = list.Find(person =>
{
    return person.IsMale && person.Age > 100;
});
```

**Visual Basic**

```
Dim found As Person = list.Find(Function(person) person.IsMale And person.Age > 100)
```

For a reference type, the default value is *null* in C# or *Nothing* in Visual Basic, and for a value type, it is 0. You could check for a default *Person* structure by adding a property such as the following.

**C#**

```
/// <summary>
/// States if the data is empty.
/// </summary>
public bool IsEmpty
{
    get
    {
        return
            (string.IsNullOrEmpty(LastName) ||
            string.IsNullOrEmpty(FirstName) ||
            Age == 0);
    }
}
```

**Visual Basic**

```
''' <summary>
''' States if the data is empty.
''' </summary>
Public ReadOnly Property IsEmpty As Boolean
    Get
        Return (String.IsNullOrEmpty(LastName) Or String.IsNullOrEmpty(FirstName) Or Age = 0)
    End Get
End Property
```

The *IsEmpty* property checks to see whether the *Age*, *Lastname*, or *FirstName* properties are *null*, which would be the case in a default structure. You could just as easily check only the *Age* property, but checking the *Lastname* and *FirstName* properties makes sure operations don't get performed on a person with no first or last name.

**Using a Method for the Search** You can also use a method to check criteria such as the following.

**C#**

```
static bool FindOver40MalesPerson person)
{
    return person.IsMale && person.Age >= 40;
}
```

**Visual Basic**

```
Function FindOver40Males(ByVal person As Person) As Boolean
    Return person.IsMale And person.Age >= 40
End Function
```

This method returns *true* if the person is male and at or over the age of 40.

Here's some code to illustrate the method.

**C#**

```
List<Person> list = new List<Person>();
list.Add(new Person("Llyina", "Julia", 12, false));
list.Add(new Person("David", "Alexander", 44, true));
list.Add(new Person("Michael", "Allen", 25, true));
list.Add(new Person("Lisa", "Andrews", 42, false));
list.Add(new Person("Jeff", "Hay", 32, true));
list.Add(new Person("Kim", "Akers", 28, false));
```

```
Person found = list.Find(FindOver40Males);
```

**Visual Basic**

```
Dim list As New List(Of Person)()
list.Add(New Person("Llyina", "Julia", 12, False))
list.Add(New Person("David", "Alexander", 44, True))
list.Add(New Person("Michael", "Allen", 25, True))
list.Add(New Person("Lista", "Andrews", 42, False))
list.Add(New Person("Jeff", "Hay", 32, True))
list.Add(New Person("Kim", "Akers", 28, False))
```

```
Dim found As Person = list.Find(AddressOf FindOver40Males)
```

This example finds a person in the list who meets the criteria in the method *FindOver40Males*. When it executes, *[“David”, “Alexander”, 44, M]* is assigned to the variable *found*. If the item is not found, the code returns a default *Person* structure.

## *FindIndex* and *FindLastIndex*

The *FindIndex* and *FindLastIndex* methods return the index of the item that matches the condition specified in the match argument. The *FindIndex* method returns the first occurrence, and the *FindLastIndex* returns the last occurrence. Both methods return a *-1* when no occurrence is found. The *Predicate(T)* method is defined as a method that takes an argument of type *T* and returns a *bool*. The method returns *true* if the argument matches the criteria and *false* otherwise.

***int FindIndex(Predicate(T) match)*** The *FindIndex* method searches the list for the first item that matches the specified condition. If the method finds an item that matches the criteria, it returns an index to that item; otherwise it returns a -1.

***int FindLastIndex(Predicate(T) match)*** The *FindLastIndex* searches the list for the last item that matches the specified condition. If the method finds an item that matches the criteria, it returns an index to that item; otherwise it returns a -1.

***Using a Method to Do the Matching*** First you need to define a method for checking the criteria. The method does not need to be static.

#### C#

```
static bool FindOver40Males (Person person)
{
    return person.IsMale && person.Age >= 40;
}
```

#### Visual Basic

```
Function FindOver40Males(ByVal person As Person) As Boolean
    Return person.IsMale And person.Age >= 40
End Function
```

Next, you need to create a driver.

#### C#

```
List<Person> list = new List<Person>();
list.Add(new Person("Llyina", "Julia", 12, false));
list.Add(new Person("David", "Alexander", 44, true));
list.Add(new Person("Michael", "Allen", 25, true));
list.Add(new Person("Lisa", "Andrews", 42, false));
list.Add(new Person("Jeff", "Hay", 42, true));
list.Add(new Person("Kim", "Akers", 28, false));

int found = list.FindIndex(FindOver40Males);

Console.WriteLine("The first male over the age of 40 is at index {0}.", found);
found = list.FindLastIndex(FindOver40Males);

Console.WriteLine("The last male over the age of 40 is at index {0}.", found);
```

## Visual Basic

```
Dim list As List(Of Person) = New List(Of Person)()
list.Add(New Person("Llyina", "Julia", 12, False))
list.Add(New Person("David", "Alexander", 44, True))
list.Add(New Person("Michael", "Allen", 25, True))
list.Add(New Person("Lisa", "Andrews", 42, False))
list.Add(New Person("Jeff", "Hay", 42, True))
list.Add(New Person("Kim", "Akers", 28, False))

Dim found As Integer = list.FindIndex(AddressOf FindOver40Males)

Console.WriteLine("The first male over the age of 40 is at index {0}.", found)

found = list.FindLastIndex(AddressOf FindOver40Males)

Console.WriteLine("The last male over the age of 40 is at index {0}.", found)
```

## Output

The first male over the age of 40 is at index 1.  
The last male over the age of 40 is at index 4.

The code looks for the first and last males over the age of 40 by calling the *FindOver40Males* method.

**Using a Lambda Expression to Do the Matching** You can use a lambda expression to find the index.

## C#

```
List<Person> list = new List<Person>();
list.Add(new Person("Llyina", "Julia", 12, false));
list.Add(new Person("David", "Alexander", 44, true));
list.Add(new Person("Michael", "Allen", 25, true));
list.Add(new Person("Lisa", "Andrews", 42, false));
list.Add(new Person("Jeff", "Hay", 42, true));
list.Add(new Person("Kim", "Akers", 28, false));

int found = list.FindIndex(person =>
{
    return person.IsMale && person.Age > 40;
});

Console.WriteLine("The first male over the age of 40 is at index {0}.", found);

found = list.FindLastIndex(person =>
{
    return person.IsMale && person.Age > 40;
});

Console.WriteLine("The last male over the age of 40 is at index {0}.", found);
```

### Visual Basic

```
Dim list As New List(Of Person())
list.Add(New Person("Llyina", "Julia", 12, False))
list.Add(New Person("David", "Alexander", 44, True))
list.Add(New Person("Michael", "Allen", 25, True))
list.Add(New Person("List", "Andrews", 42, False))
list.Add(New Person("Jeff", "Hay", 42, True))
list.Add(New Person("Kim", "Akers", 28, False))

Dim found As Integer = list.FindIndex(Function(person) person.IsMale And person.Age > 40)

Console.WriteLine("The first male over the age of 40 is at index {0}.", found)

found = list.FindLastIndex(Function(person) person.IsMale And person.Age > 40)

Console.WriteLine("The last male over the age of 40 is at index {0}.", found)
```

### Output

The first male over the age of 40 is at index 1.  
 The last male over the age of 40 is at index 4.

The code looks for the first and last males over the age of 40.

***int FindIndex(int startIndex, Predicate(T) match)*** The *FindIndex* method searches the list for the first item that matches the specified condition, starting at the specified index. If the method finds an item that matches the criteria, it returns an index to that item; otherwise it returns a -1.

***int FindLastIndex(int startIndex, Predicate(T) match)*** The *FindLastIndex* method searches the list for the last item that matches the specified condition, starting at the specified index and going back to the beginning of the list. If the method finds an item that matches the criteria, it returns an index to that item; otherwise it returns a -1.

***Using a Method to Do the Matching*** You can use a method to do the criteria check as follows.

### C#

```
List<int> list = new List<int>(new int[] { 33, 54, 23, 28, 8, 5, 14 });

int found = list.FindIndex(2, IsEven);

Console.WriteLine("The first index that contains an even number from index 2-6 is {0}.",
    found);

found = list.FindLastIndex(5, IsEven);
Console.WriteLine("The last index that contains an even number from index 0-5 is {0}.",
    found);

Console.WriteLine("An even number is located at {0}.", found);
```

### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {33, 54, 23, 28, 8, 5, 14})  
  
Dim found As Integer = list.FindIndex(2, AddressOf IsEven)  
  
Console.WriteLine("The first index that contains an even number from index 2-6 is {0}.", _  
    found)  
  
found = list.FindLastIndex(5, AddressOf IsEven)  
Console.WriteLine("The last index that contains an even number from index 0-5 is {0}.", _  
    found)  
  
Console.WriteLine("An even number is located at {0}.", found)
```

### Output

```
The first index that contains an even number from index 2-6 is 3.  
The last index that contains an even number from index 0-5 is 4.
```

The code locates the first even number in the list [33,54,23,28,8,5,14], starting at index 2, and then the last even number in the list ending at index 5.

**Using a Lambda Expression to Do the Matching** You can use a lambda expression to do the criteria check as follows.

### C#

```
List<int> list = new List<int>(new int[] { 33, 54, 23, 28, 8, 5, 14 });  
  
int found = list.FindIndex(2, item =>  
{  
    return (item % 2) == 0;  
});  
  
Console.WriteLine("The first index that contains an even number from index 2-6 is {0}.",  
    found);  
  
found = list.FindLastIndex(5, item =>  
{  
    return (item % 2) == 0;  
});  
  
Console.WriteLine("The last index that contains an even number from index 0-5 is {0}.",  
    found);
```

### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {33, 54, 23, 28, 8, 5, 14})  
  
Dim found As Integer = list.FindIndex(2, Function(item) (item Mod 2) = 0)  
  
Console.WriteLine("The first index that contains an even number from index 2-6 is {0}.", _  
    found)
```

```
found = list.FindLastIndex(5, Function(item) (item Mod 2) = 0)
Console.WriteLine("The last index that contains an even number from index 0-5 is {0}.", _
                  found)

Console.WriteLine("An even number is located at {0}.", found)
```

### Output

The first index that contains an even number from index 2-6 is 3.  
The last index that contains an even number from index 0-5 is 4.

The code locates the first even number in the list [33,54,23,28,8,5,14], starting at index 2, and then the last even number in the list ending at index 5.

***int FindIndex(int startIndex, int count, Predicate(T) match)*** The *FindIndex* method searches the list for the first item that matches the specified condition, starting at the specified index and ending at the number of items in count, which is *startIndex + count - 1*. If the method finds an item that matches the criteria, it returns the index of that item; otherwise it returns a -1.

***int FindLastIndex(int startIndex, int count, Predicate(T) match)*** The *FindLastIndex* method searches the list for the last item that matches the specified condition from (*index - count + 1*) to *index*. If the method finds an item that matches the criteria, it returns the index of that item; otherwise it returns a -1.

***Using a Method to Do the Matching*** You can use a method to do the criteria check as follows.

### C#

```
List<int> list = new List<int>(new int[] { 33, 54, 23, 28, 8, 5, 14 });

int found = list.FindIndex(2, 3, IsEven);

Console.WriteLine("The first index that contains an even number from index 2-4 is {0}.",
                  found);

found = list.FindLastIndex(5, 4, IsEven);

Console.WriteLine("The last index that contains an even number from index 2-5 is {0}.",
                  found);
```

### Output

The first index that contains an even number from index 2-4 is 3.  
The last index that contains an even number from index 2-5 is 4.

The code tries to locate the first even number in the list [33,54,23,28,8,5,14], starting at index 2, and ending at index 4, which is (*startIndex + count - 1*) and the last even number in the list from (*startIndex - count + 1*) to *startIndex*, which is 2 to 5.

**Using a Lambda to Do the Matching** You can use a lambda expression to do the criteria check as follows.

### C#

```
List<int> list = new List<int>(new int[] { 33, 54, 23, 28, 8, 5, 14 });

int found = list.FindIndex(2, 3, item =>
{
    return (item % 2) == 0;
});

Console.WriteLine("The first index that contains an even number from index 2-4 is {0}.",
    found);

found = list.FindLastIndex(5, 4, item =>
{
    return (item % 2) == 0;
});

Console.WriteLine("The last index that contains an even number from index 2-5 is {0}.",
    found);
```

### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {33, 54, 23, 28, 8, 5, 14})

Dim found As Integer = list.FindIndex(2, 3, Function(item) (item Mod 2) = 0)

Console.WriteLine("The first index that contains an even number from index 2-4 is {0}.",
    found)

found = list.FindLastIndex(5, 4, Function(item) (item Mod 2) = 0)

Console.WriteLine("The last index that contains an even number from index 2-5 is {0}.",
    found)
```

### Output

```
The first index that contains an even number from index 2-4 is 3.
The last index that contains an even number from index 2-5 is 4.
```

The code tries to locate the first even number in the list [33,54,23,28,8,5,14], starting at index 2, and ending at index 4, which is (*startindex* + *count* – 1) and the last even number in the list from (*startindex* – *count* + 1) to *startindex*, which is 2 to 5.

### *List(T) FindAll(Predicate(T) match)*

The *FindAll* method finds all items that match the condition specified in the *match* argument. The *Predicate(T)* method is defined as a method that takes an argument of type *T* and returns a *bool*. The method returns *true* when the argument matches the criteria and *false* otherwise.

**Using a Lambda Expression for the *FindAll*** A lambda expression can be used for the *FindAll* method as follows.

#### C#

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8 });
List<int> evens = list.FindAll(value => { return (value % 2) == 0; });
```

#### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8})
Dim evens As List(Of Integer) = list.FindAll(Function(value) (value Mod 2) = 0)
```

The code searches the list for all elements that are even by checking to see whether the element's *value mod 2* is equal to *0*. After execution, the variable *evens* contains the values [2,4,6,8].

**Using a Method for the *FindAll*** You can define a method to check whether an item matches specific criteria, and pass it to the *FindAll* method as shown in the following example method.

#### C#

```
static bool IsEven(int value)
{
    return (value % 2) == 0;
}
```

#### Visual Basic

```
Function IsEven(ByVal value As Integer) As Boolean
    Return ((value Mod 2) = 0)
End Function
```

The *IsEven* method checks to see whether the value is even, using the *mod* operation.

You can use the *IsEven* method with the *FindAll* method as follows.

#### C#

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8 });
List<int> evens = list.FindAll(IsEven);
```

#### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8})
Dim evens As List(Of Integer) = list.FindAll(AddressOf IsEven)
```

The code searches the list for all elements that are even by checking to see whether the element's value mod 2 is equal to 0. After execution, the variable *evens* contains the values [2,4,6,8].

## Using a Binary Search to Do the Searching

The *BinarySearch* methods use a binary search to search for the specified item. All methods require the list to be sorted before the method is called.

Some of the following examples use the *PersonComparer* class defined in the following code.

C#

```
class PersonComparer : Comparer<Person>
{
    public override int Compare(Person a, Person b)
    {
        int lastNameCompare = String.Compare(a.LastName, b.LastName, true);
        if (lastNameCompare != 0)
        {
            return lastNameCompare;
        }

        int firstNameCompare = String.Compare(a.FirstName, b.FirstName, true);
        if (firstNameCompare != 0)
        {
            return firstNameCompare;
        }

        if (a.IsMale != b.IsMale)
        {
            if (a.IsMale)
            {
                return 1;
            }
            else
            {
                return -1;
            }
        }

        if (a.Age < b.Age)
        {
            return -1;
        }

        if (a.Age > b.Age)
        {
            return 1;
        }

        return 0;
    }
}
```

### Visual Basic

```

Class PersonComparer
    Inherits Comparer(Of Person)
    Public Overrides Function Compare(ByVal a As Person, ByVal b As Person) As Integer

        Dim lastNameCompare = String.Compare(a.LastName, b.LastName, True)
        If (lastNameCompare <> 0) Then
            Return lastNameCompare
        End If

        Dim firstNameCompare = String.Compare(a.FirstName, b.FirstName, True)
        If (firstNameCompare <> 0) Then
            Return firstNameCompare
        End If

        If (a.IsMale <> b.IsMale) Then
            If (a.IsMale) Then
                Return 1
            Else
                Return -1
            End If
        End If

        If (a.Age < b.Age) Then
            Return -1
        End If

        If (a.Age > b.Age) Then
            Return 1
        End If

        Return 0
    End Function
End Class

```

**int BinarySearch(T item)** The *BinarySearch* method searches the list for the first item that matches the specified item. If the method finds an item that matches the criteria, it returns an index to that item; otherwise it returns a negative number that is the bitwise complement of the next element that is larger than *item* or the bitwise complement of *Count* if no item is larger than *item*.

### C#

```

List<int> list = new List<int>(new int[] { 33, 54, 23, 28, 8, 5, 14} );
int found = list.BinarySearch(5);
Console.WriteLine("The unsorted list is {0}", ListToString(list));
Console.WriteLine("The number 5 is located at {0} in the unsorted list.", found);
list.Sort();
Console.WriteLine("The sorted list is now {0}", ListToString(list));
found = list.BinarySearch(5);
Console.WriteLine("The number 5 is located at {0} in the sorted list.", found);

```

## Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {33, 54, 23, 28, 8, 5, 14})
Dim found As Integer = list.BinarySearch(5)

Console.WriteLine("The unsorted list is {0}", ListToString(list))
Console.WriteLine("The number 5 is located at {0} in the unsorted list.", found)
list.Sort()
Console.WriteLine("The sorted list is now {0}", ListToString(list))
found = list.BinarySearch(5)
Console.WriteLine("The number 5 is located at {0} in the sorted list.", found)
```

## Output

```
The unsorted list is [33,54,23,28,8,5,14]
The number 5 is located at -1 in the unsorted list.
The sorted list is now [5,8,14,23,28,33,54]
The number 5 is located at 0 in the sorted list.
```

Notice how the *BinarySearch* method returned a *-1* for the search on the unsorted list and a *0* for the sorted list. This is because the list needs to be sorted before the *BinarySearch* method is called.

*int BinarySearch(T item, IComparer(T) comparer)* The *BinarySearch* method searches the list for the first item that matches the specified item by using the specified comparer. If the method finds an item that matches the criteria, it returns an index to that item; otherwise it returns a negative number that is the bitwise complement of the next element that is larger than *item* or the bitwise complement of *Count* if no item is larger.

## C#

```
PersonComparer comparer = new PersonComparer();
List<Person> list = new List<Person>();
list.Add(new Person("Llyina", "Julia", 12, false));
list.Add(new Person("David", "Alexander", 44, true));
list.Add(new Person("Michael", "Allen", 25, true));
list.Add(new Person("Andrews", "Lisa", 42, false));
list.Add(new Person("Jeff", "Hay", 42, true));
list.Add(new Person("Kim", "Akers", 28, false));

Console.WriteLine("The unsorted list is {0}", ListToString(list));
list.Sort(comparer);
Console.WriteLine("The sorted list is now {0}", ListToString(list));
Person personToFind = new Person("Andrews", "Lisa", 42, false);
int found = list.BinarySearch(personToFind, comparer);
Console.WriteLine("Found {0} at {1} in the sorted list.", personToFind, found);
```

## Visual Basic

```
Dim comparer As New PersonComparer()

Dim list As New List(Of Person)()
list.Add(New Person("Llyina", "Julia", 12, False))
list.Add(New Person("David", "Alexander", 44, True))
list.Add(New Person("Michael", "Allen", 25, True))
list.Add(New Person("Andrews", "Lisa", 42, False))
```

```

list.Add(New Person("Jeff", "Hay", 42, True))
list.Add(New Person("Kim", "Akers", 28, False))

Console.WriteLine("The unsorted list is {0}", ListToString(list))
list.Sort(comparer)
Console.WriteLine("The sorted list is now {0}", ListToString(list))
Dim personToFind As Person = New Person("Andrews", "Lisa", 42, False)
Dim found As Integer = list.BinarySearch(personToFind, comparer)
Console.WriteLine("Found {0} at {1} in the sorted list.", personToFind, found)

```

### Output

The unsorted list is [[Llyina,Julia,12,F],[David,Alexander,44,M],[Michael,Allen,25,M],[Andrews,Lisa,42,F],[Jeff,Hay,42,M],[Kim,Akers,28,F]]  
 The sorted list is now [[Kim,Akers,28,F],[David,Alexander,44,M],[Michael,Allen,25,M],[Jeff,Hay,42,M],[Llyina,Julia,12,F],[Andrews,Lisa,42,F]]  
 Found [Andrews,Lisa,42,F] at 5 in the sorted list.

The code sorts the list by using the *comparer* and then tries to find *["Andrews", "Lisa", 42, F]* by using the *comparer*. The item is located at index 5 in the sorted list.

*int BinarySearch(int index, int count, T item, IComparer(T) comparer)* This overload of the *BinarySearch* method searches the list for the first item that matches the specified item by using the specified comparer and range. If the method finds an item that matches the criteria, it returns an index to that item; otherwise it returns a negative number that is the bitwise complement of the next element that is larger than *item* or the bitwise complement of *Count* if no item is larger.

### C#

```

PersonComparer comparer = new PersonComparer();
List<Person> list = new List<Person>();
list.Add(new Person("Llyina", "Julia", 12, false));
list.Add(new Person("David", "Alexander", 44, true));
list.Add(new Person("Michael", "Allen", 25, true));
list.Add(new Person("Andrews", "Lisa", 42, false));
list.Add(new Person("Jeff", "Hay", 42, true));
list.Add(new Person("Kim", "Akers", 28, false));

Console.WriteLine("The unsorted list is {0}", ListToString(list));
list.Sort(comparer);
Console.WriteLine("The sorted list is now {0}", ListToString(list));
Person personToFind = new Person("Andrews", "Lisa", 42, false);
int found = list.BinarySearch(0, 4, personToFind, comparer);
Console.WriteLine("Found {0} at {1} in the sorted list.", personToFind, found);

```

### Visual Basic

```

Dim comparer As New PersonComparer()

Dim list As New List(Of Person)()
list.Add(New Person("Llyina", "Julia", 12, False))
list.Add(New Person("David", "Alexander", 44, True))
list.Add(New Person("Michael", "Allen", 25, True))
list.Add(New Person("Andrews", "Lisa", 42, False))

```

```
list.Add(New Person("Jeff", "Hay", 42, True))
list.Add(New Person("Kim", "Akers", 28, False))

Console.WriteLine("The unsorted list is {0}", ListToString(list))
list.Sort(comparer)
Console.WriteLine("The sorted list is now {0}", ListToString(list))
Dim personToFind As Person = New Person("Andrews", "Lisa", 42, False)
Dim found As Integer = list.BinarySearch(0, 4, personToFind, comparer)
Console.WriteLine("Found {0} at {1} in the sorted list.", personToFind, found)
```

### Output

```
The unsorted list is [[Llyina,Julia,12],[David,Alexander,44,M],[Michael,Allen,25,M],[Andrews,Lisa,42,F],[Jeff,Hay,42,M],[Kim,Akers,28,F]]
The sorted list is now [[Kim,Akers,28,F],[David,Alexander,44,M],[Michael,Allen,25,M],[Jeff,Hay,42,M],[Llyina,Julia,12,F],[Andrews,Lisa,42,F]]
Found [Andrews,Lisa,42,F] at -5 in the sorted list.
```

The code sorts the list by using the *comparer* and then tries to find *["Andrews", "Lisa", 42, F]* at elements *0* through *4* using the *comparer*. Because the item isn't located, a *-5* is returned. You can find the index of the item that is greater than the item you searched for by using the following.

### C#

```
int higherIndex = ~found;
```

### Visual Basic

```
Dim higherIndex As Integer = Not found
```

## Checking the Contents of a List

Sometimes you might find it necessary to check the contents of a list. Maybe you want to see whether the list contains all even numbers, or you want to check whether the list contains an invalid phone number. You can use the *IndexOf*, *LastIndexOf*, *Contains*, *Exists*, and *TrueForAll* methods to do so.

### *int IndexOf(T item)* and *int LastIndexOf(T item)*

The *IndexOf* and *LastIndexOf* methods return the index of the item that matches the specified item. The methods search using a linear search algorithm, and they use the default equality comparer to compare items against the specified argument. The *IndexOf* method searches from the start of the list to the end of the list, and *LastIndexOf* searches the list from the end to the beginning. Both methods return a *-1* when a matching item isn't found.

**C#**

```
List<int> list = new List<int>(new int[] { 1, 2, 4, 4, 4, 6, 7 });

int firstFour = list.IndexOf(4);
int lastFour = list.LastIndexOf(4);

list[firstFour] = 3;
list[lastFour] = 5;
```

**Visual Basic**

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 4, 4, 4, 6, 7})

Dim firstFour = list.IndexOf(4)
Dim lastFour = list.LastIndexOf(4)

list(firstFour) = 3
list(lastFour) = 5
```

The code changes the list from *[1,2,4,4,4,6,7]* to *[1,2,3,4,5,6,7]*. The *IndexOf* method locates the first 4, and the *LastIndexOf* finds the last 4. The code then changes the element of the first 4 to a 3 and the element with the last 4 to a 5.

***int IndexOf(T item, int index)* and *int LastIndexOf(T item, int index)***

The *IndexOf* and *LastIndexOf* methods return the index of the item that matches the specified item within the specified range. The methods search using a linear search algorithm, and they use the default equality comparer to compare each item against the specified item. The *IndexOf* method searches from *index* to the end of the list, and the *LastIndexOf* method searches from the end of the list to *index*.

Both methods return a -1 when a matching item isn't found.

**C#**

```
List<int> list = new List<int>(new int[] { 1, 2, 7, 4, 1, 6, 7});

int extra1 = list.IndexOf(1,3);
int extra7 = list.LastIndexOf(7,3);

list[extra7] = 3;
list[extra1] = 5;
```

**Visual Basic**

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 7, 4, 1, 6, 7})

Dim extra1 = list.IndexOf(1, 3)
Dim extra7 = list.LastIndexOf(7, 3)

list(extra7) = 3
list(extra1) = 5
```

The code changes the list from [1,2,7,4,1,6,7] to [1,2,3,4,5,6,7]. The *IndexOf* method locates the first 1 after or at the item at index 3. The *LastIndexOf* finds the last 7 at or before index 3.

### ***int IndexOf(T item, int index, int count) and int LastIndexOf(T item, int index, int count)***

The *IndexOf* and *LastIndexOf* methods return the index of the item that match the specified item within the specified range. The item is searched for by using a linear search and using the default equality comparer to compare the items in the list against the specified item. The *IndexOf* method searches from the item at the position *index* for the number of items specified by *count*, and the *LastIndexOf* searches the list in reverse from the end to the number of items specified by the *count*. Both methods return a -1 when matching *item* wasn't found.

#### **C#**

```
List<int> list = new List<int>(new int[] { 1, 2, 6, 4, 2, 6, 7 });

int extra6 = list.IndexOf(6,1,6);
int extra2 = list.LastIndexOf(2,5,6);

list[extra6] = 3;
list[extra2] = 5;
```

#### **Visual Basic**

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 6, 4, 2, 6, 7})

Dim extra6 = list.IndexOf(6, 1, 6)
Dim extra2 = list.LastIndexOf(2, 5, 6)

list(extra6) = 3
list(extra2) = 5
```

The code changes the list from [1,2,6,4,2,6,7] to [1,2,3,4,5,6,7]. The *IndexOf* method locates the first 6 between the elements 1 through 6, which is *index* through (*index* + *count* - 1). The first 6 in that range is at index 2. The *LastIndexOf* finds the last 2 between the elements 5 through 0, which is *index* through (*index* - *count* + 1). The last 2 in that range is at index 5.

### ***bool Contains(T item)***

The *Contains* method checks to see whether the specified item is present in the list. When the item is in the list, it returns *true*; otherwise, it returns *false*.

#### **C#**

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 7, 6, 5, 4, 8 });

Console.WriteLine("Does the list contain a 3? {0}", list.Contains(3) ? "Yes" : "No");
Console.WriteLine("Does the list contain a 21? {0}", list.Contains(21) ? "Yes" : "No");
```

**Visual Basic**

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 7, 6, 5, 4, 8})  
  
Console.WriteLine("Does the list contain a 3? {0}", If(list.Contains(3), "Yes", "No"))  
Console.WriteLine("Does the list contain a 21? {0}", If(list.Contains(21), "Yes", "No"))
```

**Output**

```
Does the list contain a 3? Yes  
Does the list contain a 21? No
```

The method checks to see whether 3 and 21 are in the list [1,2,3,7,6,5,4,8].

***bool Exists(Predicate(T) match)***

The *Exist* method checks to see whether an item is present in the list that matches the specified criteria. Each element is checked by using a linear search. After a match is found, the search is stopped.

**Using a Method to Check for Existence** You can use the following method to do the checks.

**C#**

```
static bool IsEven(int value)  
{  
    return (value % 2) == 0;  
}
```

**Visual Basic**

```
Function IsEven(ByVal value As Integer) As Boolean  
    Return ((value Mod 2) = 0)  
End Function
```

A method may be used for the *match* parameter.

**C#**

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8});  
  
Console.WriteLine("Does the list contain an even number? {0}",  
    list.Exists(IsEven) ? "Yes" : "No");
```

**Visual Basic**

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8})  
  
Console.WriteLine("Does the list contain an even number? {0}", _  
    If(list.Exists(AddressOf IsEven), "Yes", "No"))
```

**Output**

```
Does the list contain an even number? Yes
```

The code checks to see whether an even number exists in the list [1,2,3,4,5,6,7,8] by using the *IsEven* method.

**Using a Lambda Expression to Check for Existence** You can use a lambda expression to see whether there is an element present that matches the specified criteria as follows.

#### C#

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 7, 6, 5, 4, 8 });

bool containsAnEvenNumber = list.Exists(item =>
{
    return item % 2 == 0;
});

Console.WriteLine("Does the list contain an even number? {0}",
    containsAnEvenNumber ? "Yes" : "No");
```

#### Visual Basic

```
Dim list As List(Of Integer) = New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8})

Dim containsAnEvenNumber = list.Exists(Function(item) item Mod 2 = 0)

Console.WriteLine("Does the list contain an even number? {0}", _
    If(containsAnEvenNumber, "Yes", "No"))
```

#### Output

Does the list contain an even number? Yes

The code checks to see whether an even number is present in the list [1,2,3,7,6,5,4,8] by doing a mod operation on each element until it finds one that matches the criteria.

### ***bool TrueForAll(Predicate(T) match)***

Use the *TrueForAll* method to check whether all elements in the list match the specified criteria.

**Using a Method to See Whether All Match** You can use the *TrueForAll* method to check whether all elements meet the specified criteria.

First, write a method to perform the checks.

#### C#

```
static bool IsEven(int value)
{
    return (value % 2) == 0;
}
```

### Visual Basic

```
Function IsEven(ByVal value As Integer) As Boolean
    Return ((value Mod 2) = 0)
End Function
```

The method does not have to be static.

Use the following code to use the *match* parameter.

### C#

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8});
List<int> evens = new List<int>(new int[] { 2, 4, 6, 8 });

bool allEvens = list.TrueForAll(IsEven);

Console.WriteLine("Does list contain all even numbers? {0}", allEvens ? "Yes" : "No");

allEvens = evens.TrueForAll(IsEven);

Console.WriteLine("Does evens contain all even numbers? {0}", allEvens ? "Yes" : "No");
```

### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8})
Dim evens As New List(Of Integer)(New Integer() {2, 4, 6, 8})

Dim allEvens = list.TrueForAll(AddressOf IsEven)

Console.WriteLine("Does list contain all even numbers? {0}", If(allEvens, "Yes", "No"))

allEvens = evens.TrueForAll(AddressOf IsEven)

Console.WriteLine("Does evens contain all even numbers? {0}", If(allEvens, "Yes", "No"))
```

### Output

```
Does list contain all even numbers? No
Does evens contain all even numbers? Yes
```

The code checks to see whether the list [1,2,3,4,5,6,7,8] and [2,4,6,8] contains all even numbers by using the *IsEven* method.

**Using a Lambda Expression to See Whether All Match** You can use a lambda expression to see whether all elements match specific criteria.

**C#**

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8 });
List<int> evens = new List<int>(new int[] { 2, 4, 6, 8 });

bool allEvens = list.TrueForAll(item =>
{
    return item % 2 == 0;
});

Console.WriteLine("Does list contain all even numbers? {0}", allEvens ? "Yes" : "No");

allEvens = evens.TrueForAll(item =>
{
    return item % 2 == 0;
});

Console.WriteLine("Does evens contain all even numbers? {0}", allEvens ? "Yes" : "No");
```

**Visual Basic**

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8})
Dim evens As New List(Of Integer)(New Integer() {2, 4, 6, 8})

Dim allEvens = list.TrueForAll(Function(item) (item Mod 2) = 0)

Console.WriteLine("Does list contain all even numbers? {0}", If(allEvens, "Yes", "No"))

allEvens = evens.TrueForAll(Function(item) (item Mod 2) = 0)

Console.WriteLine("Does evens contain all even numbers? {0}", If(allEvens, "Yes", "No"))
```

**Output**

```
Does list contain all even numbers? No
Does evens contain all even numbers? Yes
```

The code checks to see whether the list [1,2,3,4,5,6,7,8] and [2,4,6,8] contain all even numbers by using the *mod* operation.

## Modifying the List

Use the *ConvertAll*, *Reverse*, and *Trim* methods to modify the list.

### ***List(TOutput) ConvertAll(Converter(T, TOutput) converter)***

The *ConvertAll* method converts all items in the list to another type. It returns a list containing all the converted elements. This can be useful when you want to convert a list of integers to strings or vice versa. *Converter(T, TOutput)* is defined as follows.

**C#**

```
delegate TOutput Converter<TInput, TOutput>(TInput input);
```

**Visual Basic**

```
Public Delegate Function Converter(Of TInput,TOutput) (TInput) As TOutput
```

The method takes an argument as type *T* and returns an object of type *TOutput*.

**Using a Method to Do the Conversion** You can implement the *Converter* function as a method. First, define a method, as shown in the following code.

**C#**

```
static string UnpackPhoneNumber(long phonenumber)
{
    long areacode, exccode, subnum;

    // Get the first 3 digits of the phonenumber
    areacode = Math.Floor(phonenumber / 10000000);

    // Change the phonenumber variable to the last 7 digits
    Math.DivRem(phonenumber, 10000000, out phonenumber);

    // Get the exchange code or first 3 digits
    exccode = phonenumber / 10000;

    // Get the subscriber number from the last 4 digits
    Math.DivRem(phonenumber, 1000, out subnum);

    return string.Format("{0:D3} {1:D3}-{2:D4}", areacode, exccode, subnum);
}
```

**Visual Basic**

```
Function UnpackPhoneNumber(ByVal phonenumber As Long) As String
    Dim areacode As Long, exccode As Long, subnum As Long

    ' Get the first 3 digits of the phonenumber
    areacode = Math.Floor(phonenumber / 10000000)

    ' Change the phonenumber variable to the last 7 digits
    Math.DivRem(phonenumber, 10000000, phonenumber)

    ' Get the exchange code or first 3 digits
    exccode = Math.Floor(phonenumber / 10000)

    ' Get the subscriber number from the last 4 digits
    Math.DivRem(phonenumber, 1000, subnum)

    Return String.Format("{0:D3} {1:D3}-{2:D4}", areacode, exccode, subnum)
End Function
```

This *UnpackPhoneNumber* method converts a phone number stored as an 8-byte *long* to a *string*. Because this is only an example, error checking has been omitted. Also, the method doesn't need to be static.

The following code can be used as a driver.

#### C#

```
List<long> list = new List<long>(new long[] { 9015550100, 9015550188, 9015550113 });

List<string> phonenumbers = list.ConvertAll<string>( UnpackPhoneNumber);
```

#### Visual Basic

```
Dim list As New List(Of Long)(New Long() {9015550100, 9015550188, 9015550113})

Dim phonenumbers = list.ConvertAll(Of String)(AddressOf UnpackPhoneNumber)
```

The code turns the list [9015550100, 9015550188, 9015550113] into [(901) 555-0100,(901) 555-0188,(901) 555-0113], expanding each phone number from an 8-byte *long* to a 14-character *string*.

**Using a Lambda Expression to Do the Conversion** You can also use a lambda expression to perform the conversion, as follows.

#### C#

```
List<long> list = new List<long>(new long[] { 9015550100, 9015550188, 9015550113 });

List<string> phonenumbers = list.ConvertAll<string>(item =>
{
    long areaCode = item / 10000000;
    item -= (areaCode * 10000000);
    long exccode = item / 10000;

    return string.Format("{0:D3} {1:D3}-{2:D4}", areaCode, exccode,
                           item - (exccode * 10000));
});
```

#### Visual Basic 2010

```
Dim list As New List(Of Long)(New Long() {9015550100, 9015550188, 9015550113})

Dim phonenumbers = list.ConvertAll(Of String)(Function(item)
    Dim areaCode As Long = item / 10000000
    item -= (areaCode * 10000000)
    Dim exccode As Long = item / 10000

    Return String.Format("{0:D3} {1:D3}-{2:D4}", areaCode, exccode,
                           item - (exccode * 10000))
End Function)
```

The preceding code turns the list [9015550100, 9015550188, 9015550113] into [(901) 555-0100,(901) 555-0188,(901) 555-0113] by using some math to parse the phone numbers.

### **void Reverse()**

The *Reverse* method reverses the order of the elements in the list, using the *Array.Reverse* method.

#### C#

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8 });

list.Reverse();
```

#### Visual Basic

```
Dim list As New List(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8})

list.Reverse()
```

The list is changed from [1,2,3,4,5,6,7,8] to [8,7,6,5,4,3,2,1] after the *Reverse* method is called.

### **void Reverse(int index, int count)**

This *Reverse* method overload reverses the elements in the list at the specified range. It performs the operation by using the *Array.Reverse* method.

#### C#

```
List<int> list = new List<int>(new int[] { 1, 2, 3, 7, 6, 5, 4, 8 });

list.Reverse(3, 4);
```

#### Visual Basic

```
Dim list As List(Of Integer) = New List(Of Integer)(New Integer() {1, 2, 3, 7, 6, 5, 4, 8})

list.Reverse(3, 4)
```

The list is changed from [1,2,3,7,6,5,4,8] to [1,2,3,4,5,6,7,8] after calling the *Reverse* method.

### **void TrimExcess()**

The *TrimExcess* method changes the capacity of the list when the list's *Count* is less than 90 percent of the current *Capacity*. You need to call both *Clear* and then *TrimExcess* to restore a list to a condition similar to the original list created with the default constructor.

## **LinkedList(T) Overview**

The *LinkedList(T)* class is a generic implementation of a doubly linked list. This class uses *LinkedListNode(T)* for the nodes inside of the linked list. The *LinkedList(T)* class has no non-generic equivalent.



**More Info** See Chapter 1 for more information about doubly linked lists.

Adds, removals, and insertions are  $O(1)$  operations.

## Using the *LinkedList(T)* Class

You should refer to the “Doubly Linked List Implementation” section in Chapter 1 to see when to use the *LinkedList(T)* class, but in general, you use the class when you plan to do a lot of inserts and removals. Items in the list are represented in this book by using the notation  $[E_0 E_1 E_2 E_3 \dots E_N]$ , where  $E_0$  is node 0,  $E_1$  is the next node, and so on.

### Creating a New *LinkedList(T)*

A new linked list can be created using either the *LinkedList(T)()* or the *LinkedList(T)(IEnumerable(T) collection)* constructor.

#### *LinkedList(T)()*

This constructor creates a new empty linked list.

##### C#

```
LinkedList<String> llist = new LinkedList<String>();
```

##### Visual Basic

```
Dim llist As New LinkedList(Of String)()
```

The *llist* variable contains a new empty *LinkedList<String>* object.

#### *LinkedList(T)(IEnumerable(T) collection)*

This constructor creates a new linked list with the items in parameter *collection* as the items.

### Adding to the *LinkedList(T)*

Nodes can be added to the linked list with the *AddAfter*, *AddBefore*, *AddFirst*, and *AddLast* methods. Values *can* be duplicated in the linked list.

#### **void AddAfter(LinkedListNode(T) node, LinkedListNode(T) newNode)**

The *AddAfter* method adds the *newNode* after the specified *node*. You can obtain one of the list’s *LinkedListNode(T)* nodes from the *Add* or *Find* methods or the *First* or *Last* properties.

**C#**

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "like" });

llist.AddAfter(llist.Last, new LinkedListNode<string>("dogs"));
```

**Visual Basic**

```
Dim llist As New LinkedList(Of String)(New String() {"I", "like"})

llist.AddAfter(llist.Last, New LinkedListNode(Of String)("dogs"))
```

The code changes the list from `["I", "like"]` to `["I", "like", "dogs"]`. The node containing `"dogs"` is added after the `Last` node.

***LinkedListNode(T) AddAfter(LinkedListNode(T) node, T value)***

The `AddAfter` method adds the specified value after `node`. A new `LinkedListNode(T)` is created to hold the value and then returned. You can obtain one of the list's `LinkedListNode(T)` nodes from the `Add` or `Find` methods or the `First` or `Last` properties.

**C#**

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "like" });

llist.AddAfter(llist.Last, "dogs");
```

**Visual Basic**

```
Dim llist New LinkedList(Of String)(New String() {"I", "like"})

llist.AddAfter(llist.Last, "dogs")
```

The code changes the list from `["I", "like"]` to `["I", "like", "dogs"]`. The node containing `"dogs"` is added after the `Last` node.

***void AddBefore(LinkedListNode(T) node, LinkedListNode(T) newNode)***

The `AddBefore` method adds the `newNode` before `node`. You can obtain one of the list's `LinkedListNode(T)` nodes from the `Add` or `Find` methods or the `First` or `Last` properties.

**C#**

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "like", "dogs" });

llist.AddBefore(llist.First, new LinkedListNode<string>("I"));
```

**Visual Basic**

```
Dim llist As New LinkedList(Of String)(New String() {"like", "dogs"})

llist.AddBefore(llist.First, New LinkedListNode(Of String)("I"))
```

The code changes the list from `["like", "dogs"]` to `["I", "like", "dogs"]`. The node containing `"I"` is added before the `First` node.

### ***LinkedListNode(T) AddBefore(LinkedListNode(T) node, T value)***

The *AddBefore* method adds the specified value before the specified node. A new *LinkedListNode(T)* is created to hold the value and then returned. You can obtain one of the list's *LinkedListNode(T)* nodes from the *Add* or *Find* methods or the *First* or *Last* properties.

#### C#

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "like", "dogs" });

LinkedListNode<string> foundNode = llist.Find("like");

llist.AddBefore(foundNode, "I");
```

#### Visual Basic

```
Dim llist As New LinkedList(Of String)(New String() {"like", "dogs"})

Dim foundNode As LinkedListNode(Of String) = llist.Find("like")

llist.AddBefore(foundNode, "I")
```

The code changes the list from *["like", "dogs"]* to *["I", "like", "dogs"]*. The node containing *"I"* is added before the found node. The *Find* method, which is discussed later in this chapter, is used to find the node to add the newly created node in front of.

### ***LinkedListNode(T) AddFirst(T item)***

The *AddFirst* method adds an item to the beginning of the list. The node that was created to hold that item is passed back.

#### C#

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "like", "dogs" });

llist.AddFirst("I");
```

#### Visual Basic

```
Dim llist As New LinkedList(Of String)(New String() {"like", "dogs"})

llist.AddFirst("I")
```

The code changes the list from *["like", "dogs"]* to *["I", "like", "dogs"]*.

### ***void AddFirst(LinkedListNode(T) node)***

The *AddFirst* method adds the specified node to the beginning of the list. An *ArgumentNullException* is thrown if a *null* value is passed in, or an *InvalidOperationException* is thrown if the specified node belongs to another list.

**C#**

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "like", "dogs" });

llist.AddFirst(new LinkedListNode<string>("I"));
```

**Visual Basic**

```
Dim llist As New LinkedList(Of String)(New String() {"like", "dogs"})

llist.AddFirst(New LinkedListNode(Of String)("I"))
```

The code changes the list from `["like", "dogs"]` to `["I", "like", "dogs"]`. The `LinkListNode(T)` object is created with the value `"I"` and added to the beginning of the list.

### ***LinkedListNode(T) AddLast (T item)***

The `AddLast` method adds an item to the end of the list. The node that was created to hold that item will be passed back.

**C#**

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "like" });

llist.AddLast("dogs");
```

**Visual Basic**

```
Dim llist As New LinkedList(Of String)(New String() {"I", "like"})

llist.AddLast("dogs")
```

The code changes the list from `["I", "like"]` to `["I", "like", "dogs"]`.

### ***void AddLast (LinkedListNode(T) node)***

The `AddLast` method adds the specified node to the end of the list. An `ArgumentNullException` is thrown if a `null` value is passed in, or an `InvalidOperationException` is thrown if the specified node belongs to another list.

**C#**

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "like" });

llist.AddLast(new LinkedListNode<string>("dogs"));
```

**Visual Basic**

```
Dim llist As New LinkedList(Of String)(New String() {"I", "like"})

llist.AddLast(New LinkedListNode(Of String)("dogs"))
```

The code changes the list from `["I", "like"]` to `["I", "like", "dogs"]`. The `LinkListNode(T)` object is created with the value `"dogs"` and added to the end of the list.

## Removing Nodes from the *LinkedList(T)*

Nodes can be removed from the linked list with the *Remove*, *Clear*, *RemoveFirst*, and *RemoveLast* methods.

### ***void Clear()***

The *Clear* method removes all nodes from the list. All references in the nodes are released.

#### C#

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "love", "dogs" });

llist.Clear();
```

#### Visual Basic

```
Dim llist As New LinkedList(Of String)(New String() {"I", "love", "dogs"})

llist.Clear()
```

The linked list does not contain any elements after the *Clear* method is called.

### ***bool Remove(T value)***

The *Remove* method removes the specified value from the list. The method performs a linear search to locate the specified item. The method returns *true* when the item is found and removed from the list; otherwise, it returns *false*.

#### C#

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "don't", "like",
    "dogs" });

llist.Remove("don't");
```

#### Visual Basic

```
Dim llist As New LinkedList(Of String)(New String() {"I", "don't", "like", "dogs"})

llist.Remove("don't")
```

The code changes the list from `["I","don't","like","dogs"]` to `["I","like","dogs"]` after the *Remove* method is called.

### ***bool Remove(LinkedListNode(T) node)***

The *Remove* method removes the specified node from the list. The method throws an *InvalidOperationException* if the node isn't present in the linked list, and an *ArgumentNullException* if the specified node is *null*. You can obtain a *LinkedListNode(T)* from one of the *Add* or *Find* methods or the *First* or *Last* properties.

**C#**

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "don't", "like",  
    "dogs" });  
  
LinkedListNode<string> removeNode = llist.Find("don't");  
llist.Remove(removeNode);
```

**Visual Basic**

```
Dim llist As New LinkedList(Of String)(New String() {"I", "don't", "like", "dogs"})  
  
Dim removeNode As LinkedListNode(Of String) = llist.Find("don't")  
llist.Remove(removeNode)
```

The code changes the list from `["I","don't","like","dogs"]` to `["I","like","dogs"]` after the *Remove* method is called. You can use the *Find* method to find the node you want to remove if you do not already have it.

***void RemoveFirst()***

The *RemoveFirst* method removes the first node of the list. An *InvalidOperationException* is thrown if you call this method on an empty list.

**C#**

```
LinkedList<int> llist = new LinkedList<int>(new int[] { 1, 1, 2, 3, 4 });  
  
llist.RemoveFirst();
```

**Visual Basic**

```
Dim llist As New LinkedList(Of Integer)(New Integer() {1, 1, 2, 3, 4})  
  
llist.RemoveFirst()
```

The list is changed from `[1,1,2,3,4]` to `[1,2,3,4]` when the *RemoveFirst* method is called.

***void RemoveLast()***

The *RemoveLast* method removes the last node of the list. An *InvalidOperationException* is thrown if you call this method on an empty list.

**C#**

```
LinkedList<int> llist = new LinkedList<int>(new int[] { 1, 2, 3, 4, 4 });  
  
llist.RemoveLast();
```

**Visual Basic**

```
Dim llist As New LinkedList(Of Integer)(New Integer() {1, 2, 3, 4, 4})  
  
llist.RemoveLast()
```

The list is changed from `[1,2,3,4,4]` to `[1,2,3,4]` when the *RemoveLast* method is called.

## Obtaining Information on the *LinkedList(T)*

Sometimes you need to retrieve information about the linked list without altering the contents of it. The *Find*, *FindLast*, and *Contains* methods and *First*, *Last*, and *Count* properties allow you to do so.

### *LinkedListNode(T) Find(T value)* and *LinkedListNode(T) FindLast(T value)*

The *Find* and *FindLast* methods return the node that contains the specified value. The list is traversed using a linear search. The *Find* method locates the first node, and the *FindLast* method locates the last node. Both methods return the node that contains the specified value; otherwise they return *null*.

#### C#

```
LinkedList<int> llist = new LinkedList<int>(new int[] { 1, 2, 4, 4, 4, 6 });

Print(llist.ToArray());

llist.Find(4).Value = 3;
llist.FindLast(4).Value = 5;

Print(llist.ToArray());
```

#### Visual Basic

```
Dim llist As New LinkedList(Of Integer)(New Integer() {1, 2, 4, 4, 4, 6})

Print(llist.ToArray())

llist.Find(4).Value = 3
llist.FindLast(4).Value = 5

Print(llist.ToArray())
```

#### Output

```
[1,2,4,4,4,6]
[1,2,3,4,5,6]
```

The code changes the linked list from [1,2,4,4,4,6] to [1,2,3,4,5,6]. It does so by locating the first 4 in the group and changing it to 3 and then finding the last 4 in the group and changing it to 5.

***LinkedListNode(T) First { get; } and LinkedListNode(T) Last { get; }***

The *First* and *Last* properties obtain the first and last node in the list respectively. Both properties return *null* if the list is empty.

**C#**

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "love", "dogs" });

Console.WriteLine(llist.First.Value);
Console.WriteLine(llist.Last.Value);
```

**Visual Basic**

```
Dim llist As New LinkedList(Of String)(New String() {"I", "love", "dogs"})

Console.WriteLine(llist.First.Value)
Console.WriteLine(llist.Last.Value)
```

**Output**

```
I
dogs
```

*Console.WriteLine(llist.First.Value)* causes “I” to be written to the console because it is the first one in the list. *Console.WriteLine(llist.Last.Value)* causes “dogs” to be written to the console because it is the last one in the list.

***int Count { get; }***

The *Count* property returns the number of nodes in the list.

**C#**

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "like", "dogs" });

Console.WriteLine("The list contains {0} nodes.", llist.Count);
```

**Visual Basic**

```
Dim llist As New LinkedList(Of String)(New String() {"I", "love", "dogs"})

Console.WriteLine("The list contains {0} nodes.", llist.Count)
```

**Output**

```
The list contains 3 nodes.
```

The output shows the list as having 3 nodes because it contains [“I”, “like”, “dogs”].

## **bool Contains(*T* value)**

The *Contains* method can be used to determine whether a value is present in the list. It returns *true* if the value is in the list; otherwise it returns *false*.

### C#

```
LinkedList<string> llist = new LinkedList<string>(new string[] { "I", "like", "dogs" });

Console.WriteLine("Does the list contain \"dogs\"? {0}",
    llist.Contains("dogs") ? "Yes" : "No");
Console.WriteLine("Does the list contain \"cats\"? {0}",
    llist.Contains("cats") ? "Yes" : "No");
```

### Visual Basic

```
Dim llist As LinkedList(Of String) = New LinkedList(Of String)(New String() {"I", "love",
"dogs"})

Console.WriteLine("Does the list contain ""dogs""? {0}",
    If(llist.Contains("dogs"), "Yes", "No"))
Console.WriteLine("Does the list contain ""cats""? {0}",
    If(llist.Contains("cats"), "Yes", "No"))
```

### Output

```
Does the list contain "dogs"? Yes
Does the list contain "cats"? No
```

The code checks to see whether *"dogs"* and *"cats"* are in the list.

## **Summary**

In this chapter, you learned about the equality and ordering comparers. Understanding the comparers helps you understand how the .NET Framework compares and orders objects. You also learned about the built-in array class called *List(*T*)*. You learned how to use it and how it differs from the nongeneric implementation called *ArrayList*. And finally, you learned how to create a generic linked list by using the *LinkedList(*T*)* class.



## Chapter 5

# Generic and Support Collections

After completing this chapter, you will be able to

- Use the *Queue(T)* class.
- Use the *Stack(T)* class.
- Use the *Dictionary(TKey, TValue)* class.
- Use the *BitArray* class.
- Use the *CollectionBase* and *DictionaryBase* classes.
- Use the *HashSet(T)* class.
- Use *SortedList(TKey,TValue)* and *SortedDictionary(TKey,TValue)* classes.

## **Queue(*T*) Overview**

The *Queue(T)* class is a generic implementation of a First In, First Out (FIFO) collection or queue, as discussed in Chapter 3, "Understanding Collections: Queues, Stacks, and Circular Buffers."

The *Queue(T)* class uses an array for data storage in the same way that the *QueuedArray(T)* class does in Chapter 3

Pushing items onto a queue that has full capacity takes  $O(n)$  operations; otherwise it takes only  $O(1)$  operations. Popping an item from the queue always take  $O(1)$  operations.

The *Queue(T)* class is the generic version of the *Queue* class, so it does not have to box value types like the nongeneric *Queue* class must do.

## **Using the Queue(*T*) Class**

You can refer to the "Uses of Queues" section in Chapter 3 regarding when to use the *Queue(T)* class, but in general, you use the class when you will always access items in the order you receive them. Because the *Queue(T)* class is the preferred way of creating a queue, the *Queue* class is not discussed in this book. Items in the queue are represented in the book by using the notation  $[E_0, E_1, E_2, E_3 \dots E_N]$ , where  $E_0$  is the first item in the queue,  $E_1$  is the second, and so on.

## Creating a Queue

You can create a queue by using one of the three constructors described in the following sections.

### ***Queue(T)()***

The default constructor can be used to create an empty queue as follows.

#### C#

```
Queue<int> queue = new Queue<int>();
```

#### Visual Basic

```
Dim queue As New Queue(Of Integer)()
```

The created queue contains no items.

### ***Queue(T)(IEnumerable(T))***

You can create a queue from another collection that contains the items you want to copy to the queue. Each item is added to the queue in the order it is enumerated.

#### C#

```
int[] values = new int[] { 34, 2, 1, 88, 53 };
Queue<int> queue = new Queue<int>(values);
```

#### Visual Basic

```
Dim values As New Integer() {34, 2, 1, 88, 53}
Dim queue As New Queue(Of Integer)(values)
```

After the code is executed, *queue* contains the values [34, 2, 1, 88, 53].

### ***Queue(T)(int size)***

You can create a queue with an initial capacity.

#### C#

```
Queue<int> queue = new Queue<int>(100);
```

#### Visual Basic

```
Dim queue As New Queue(Of Integer)(100)
```

After the code is executed, *queue* contains an empty queue with a capacity of 100.

## Adding Items to the Queue

Adding items to a queue is referred to as *pushing an item onto the queue*. You can do this by using the *Enqueue* method.

### **void Enqueue(T item)**

Items can be added to the end of the queue through the *Enqueue* method. The *Enqueue* method takes as an argument the item you want to add to the end of the queue.

#### C#

```
Queue<int> queue = new Queue<int>(new int [] { 22, 3, 6, 19 });

queue.Enqueue(33);
```

#### Visual Basic

```
Dim queue As New Queue(Of Integer)(New Integer() {22, 3, 6, 19})

queue.Enqueue(33)
```

The queue is changed from [22,3,6,19] to [22,3,6,19,33] after the *Enqueue* method is called with the item 33.

## Removing Items from the Queue

Items can be removed from the queue by using the *Clear* and *Dequeue* methods. It is important to remember that the capacity of the queue does not change when you remove items from the queue. You need to call the *TrimExcess* method to change the capacity of the queue.

### **void Clear()**

All items are removed when the *Clear* method is called.

#### C#

```
Queue<int> queue = new Queue<int>(new int [] { 22, 3, 6, 19 });

queue.Clear();
```

#### Visual Basic

```
Dim queue As New Queue(Of Integer)(New Integer() {22, 3, 6, 19})

queue.Clear()
```

The queue is changed from [22,3,6,19] to [] after the *Clear* method is called.

### T Dequeue()

An item can be removed from the queue by calling the *Dequeue* method. The *Dequeue* method always removes the first item from the queue.

#### C#

```
Queue<int> queue = new Queue<int>(new int [] { 22, 3, 6, 19 });

Console.WriteLine(queue.Dequeue());
```

#### Visual Basic

```
Dim queue As New Queue(Of Integer)(New Integer() {22, 3, 6, 19})

Console.WriteLine(queue.Dequeue())
```

#### Output

```
22
```

The queue changes from [22,3,6,19] to [3,6,19] after the *Dequeue* method is called. That is because item 22 is removed from the list and written to the console.

## Checking the Queue

Sometimes you need to check the contents of the queue. With the *Peek* and *Contains* methods, you can check the contents of the queue without affecting it.

### T Peek()

You can use the *Peek* method when you want to view the first element in the queue without changing the contents of it.

#### C#

```
Queue<int> queue = new Queue<int>(new int [] { 22, 3, 6, 19 });

Console.WriteLine(queue.Peek());
```

#### Visual Basic

```
Dim queue As New Queue(Of Integer)(New Integer() {22, 3, 6, 19})

Console.WriteLine(queue.Peek())
```

#### Output

```
22
```

The queue remains [22,3,6,19] after the *Peek* method is called. The first element in the queue, 22, is written to the console. This allows you to view the item you are going to remove without removing it.

## **bool Contains(*T item*)**

You can use the *Contains* method to check the contents of the queue for a specific item. When the item exists in the queue, it returns *true*; otherwise it returns *false*.

### C#

```
Queue<int> queue = new Queue<int>(new int [] { 22, 3, 6, 19 });

Console.WriteLine("Does the queue contain a 6? {0}", queue.Contains(6) ? "Yes" : "No");
Console.WriteLine("Does the queue contain a 5? {0}", queue.Contains(5) ? "Yes" : "No");
```

### Visual Basic

```
Dim queue As New Queue(Of Integer)(New Integer() {22, 3, 6, 19})

Console.WriteLine("Does the queue contain a 6? {0}", If(queue.Contains(6), "Yes", "No"))
Console.WriteLine("Does the queue contain a 5? {0}", If(queue.Contains(5), "Yes", "No"))
```

### Output

```
Does the queue contain a 6? Yes
Does the queue contain a 5? No
```

The code checks to see whether 6 and 5 are contained in the queue [22, 3, 6, 19].

## ***int Count***

The *Count* property returns the number of items currently in the queue.

### C#

```
Queue<int> queue = new Queue<int>(new int[] { 22, 3, 6, 19 });

Console.WriteLine("The queue contains {0} items.", queue.Count);
```

### Visual Basic

```
Dim queue As New Queue(Of Integer)(New Integer() {22, 3, 6, 19})

Console.WriteLine("The queue contains {0} items.", queue.Count)
```

### Output

```
The queue contains 4 items.
```

The *Count* is equal to 4 because the stack contains the elements [22, 3, 6, 19].

## **Cleaning the Queue**

Sometimes you may find it useful to free memory used by the queue. The *TrimExcess* method helps you reclaim valuable memory.

### **void TrimExcess()**

The *TrimExcess* method reduces the capacity of the queue to the actual *Count* size. It is important to note that the method only executes if the *Count* is less than 90 percent of the *Capacity*.

## **Stack(*T*) Overview**

The *Stack(*T*)* class is a generic implementation of a Last In, First Out (LIFO) collection or a stack as discussed in Chapter 3. This class uses an array for its internal data storage.



**More Info** For more information about how arrays work, see Chapter 1, “Understanding Collections: Arrays and Linked Lists.”

Pushing onto a stack that has full capacity takes  $O(n)$  operations; otherwise it takes only  $O(1)$  operations. Popping always takes  $O(1)$  operations.

The *Stack(*T*)* class is the generic version of the *Stack* class, so it does not have to box value types like the *Stack* class has to do.

## **Using the Stack(*T*) Class**

You can refer to the “Uses of Stacks” section in Chapter 3 regarding when to use the *Stack(*T*)* class, but in general, you use the class when you always need to access the last item you received first. Because the *Stack(*T*)* class is the preferred way of creating a stack, this book does not discuss the nongeneric *Stack* class. This book represents items in the stack by using the notation  $[E_N \dots, E_3, E_2, E_1, E_0]$ , where  $E_0$  is the first item to be removed,  $E_1$  is the second, and so on.

### **Creating a Stack**

You can create a stack by using one of the three constructors described in the following sections.

#### **Stack(*T*)()**

The default constructor can be used to create an empty stack as follows.

##### **C#**

```
Stack<int> stack = new Stack<int>();
```

##### **Visual Basic**

```
Dim stack As New Stack(Of Integer)()
```

## Stack (*T*)(*IEnumerable*(*T*))

You can also create a stack from another collection that contains the items you want to copy to the stack.

### C#

```
int[] values = new int[] { 34, 2, 1, 88, 53 };
Stack<int> stack = new Stack<int>(values);
```

### Visual Basic

```
Dim values As New Integer() {34, 2, 1, 88, 53}
Dim stack As New Stack(Of Integer)(values)
```

After the code is executed, *stack* contains the values [34,2,1,88,53].

## Stack (*T*)(*int size*)

You can also create a stack with an initial capacity.

### C#

```
Stack<int> stack = new Stack<int>(100);
```

### Visual Basic

```
Dim stack As New Stack(Of Integer)(100)
```

After the code is executed, *stack* contains an empty stack with a capacity of 100.

## Adding Items to the Stack

You can use the *Push* method when you need to add items to the top of the stack.

### void Push(*T item*)

The *Push* method lets you add an item to the top of the stack.

### C#

```
Stack<int> stack = new Stack<int>(new int[] { 6, 87, 13, 29, 7 });
stack.Push(46);
```

### Visual Basic

```
Dim stack As New Stack(Of Integer)(New Integer() {6, 87, 13, 29, 7})
stack.Push(46)
```

The stack is changed from [6,87,13,29,7] to [6,87,13,29,7,46] after the *Push* method is called. This is because the *Push* method adds the item, 46, to the top of the stack.

## Removing Items from the Stack

The *Clear* and *Pop* methods can be used to remove items from the stack. It is important to remember that the capacity of the stack does not change. You need to call the *TrimExcess* method to change the capacity of the stack.

### **void Clear()**

All items can be removed from the stack by using the *Clear* method.

#### C#

```
Stack<int> stack = new Stack<int>(new int[] { 6, 87, 13, 29, 7 });

stack.Clear();
```

#### Visual Basic

```
Dim stack As New Stack(Of Integer)(New Integer() {6, 87, 13, 29, 7})

stack.Clear()
```

The stack is changed from [6,87,13,29,7] to [] after the *Clear* method is called. This is because the *Clear* method removes all items from the stack.

### **T Pop()**

The *Pop* method can be used to remove the item from the top of the stack. The top of the stack is always the last item added.

#### C#

```
Stack<int> stack = new Stack<int>(new int[] { 6, 87, 13, 29, 7 });

Console.WriteLine(stack.Pop());
```

#### Visual Basic

```
Dim stack As New Stack(Of Integer)(New Integer() {6, 87, 13, 29, 7})

Console.WriteLine(stack.Pop())
```

#### Output

7

The stack is changed from [6,87,13,29,7] to [6,87,13,29] after the *Pop* method is called. This is because the *Pop* method removes the last item in the stack.

## Checking the Stack

You can use the *Peek* and *Contains* methods to check the contents of the stack without altering it.

### **T Peek()**

The *Peek* method can be used to view the item at the top of the stack. The item at the top of the stack is returned but not removed.

#### C#

```
Stack<int> stack = new Stack<int>(new int[] { 6, 87, 13, 29, 7 });

Console.WriteLine(stack.Peek());
```

#### Visual Basic

```
Dim stack As New Stack(Of Integer)(New Integer() {6, 87, 13, 29, 7})

Console.WriteLine(stack.Peek())
```

#### Output

```
7
```

The stack remains [6,87,13,29,7] after the *Peek* method is called because it doesn't alter the stack. In this case, a 7 is returned from the *Peek* method.

### **bool Contains(T item)**

You can see whether an item exists in the stack by calling the *Contains* method. The item searches linearly to find the item by using the default equality comparer. It returns *true* when the item exists in the stack; otherwise it returns *false*.

#### C#

```
Stack<int> stack = new Stack<int>(new int[] { 6, 87, 13, 29, 7 });

Console.WriteLine("Does the stack contain a 13? {0}", stack.Contains(13) ? "Yes" : "No");
Console.WriteLine("Does the stack contain a 12? {0}", stack.Contains(12) ? "Yes" : "No");
```

#### Visual Basic

```
Dim stack As New Stack(Of Integer)(New Integer() {6, 87, 13, 29, 7})

Console.WriteLine("Does the stack contain a 13? {0}", If(stack.Contains(13), "Yes", "No"))
Console.WriteLine("Does the stack contain a 12? {0}", If(stack.Contains(12), "Yes", "No"))
```

#### Output

```
Does the stack contain a 13? Yes
Does the stack contain a 12? No
```

The code checks to see whether 13 and 12 exist in the stack [6,87,13,29,7].

### ***int Count***

The *Count* property returns the number of items currently in the stack.

#### C#

```
Stack<int> stack = new Stack<int>(new int[] { 6, 87, 13, 29, 7 });

Console.WriteLine("The stack contains {0} items.", stack.Count);
```

#### Visual Basic

```
Dim stack As New Stack(Of Integer)(New Integer() {6, 87, 13, 29, 7})

Console.WriteLine("The stack contains {0} items.", stack.Count)
```

#### Output

The stack contains 5 items.

The *Count* is equal to 5 because the stack contains the elements [7,29,13,87,6].

## **Cleaning the Stack**

The *TrimExcess* method can be used to reclaim space.

### ***TrimExcess***

The *TrimExcess* method reduces the capacity of the stack to the actual *Count* size. It is important to note that the method executes only if the *Count* is less than 90 percent of the *Capacity*.

## ***Dictionary(TKey,TValue)* Overview**

The *Dictionary(TKey,TValue)* class is a generic associative array that uses a hash table for its internal implementation. You learned about associative arrays in Chapter 2, "Understanding Collections: Associative Arrays." The *Hashtable* class is not used to implement the hash table used by the *Dictionary(TKey, TValue)* class.

Adding to a dictionary that is full takes  $O(n)$  operations; otherwise it takes only  $O(1)$  operations. Removing approaches an  $O(1)$  operation.

The *Dictionary(T)* class is the generic version of the *Hashtable* class, so it does not have to box value types like the *Hashtable* class has to do.

## Understanding *Dictionary(TKey,TValue)* Implementation

When an item is added to the hash table, the hash code is retrieved and the sign is removed from the hash code. Next, the bucket index is determined by doing a modulus operation on the hash code with the number of buckets in the hash table. Each bucket references a linked list to solve collisions. The item is then added to the beginning of the linked list by retrieving a node from the freed linked list. The table is resized if there are no nodes left in the freed linked list. Resizing requires that a new bucket array is created and all items are removed from the old bucket array and then added to the new bucket array; this is all done internally. This is done because the number of buckets is used to determine the bucket index of an item and a resize changes the number of buckets.

With a lookup, the bucket is determined by using the same procedure that is used with adding an item. After the bucket is determined, the linked list associated with that bucket is accessed and then traversed to find the item you are looking for. Each node contains the hash code of the item that is stored in it. The hash code is first compared against the hash code of the item you are looking at for performance reasons. It is faster to do an integer comparison than the majority of other data comparisons. If that hash code comparison passes, the key is then compared and the procedure continues to the next node if the key comparison fails.

With a remove, the item is determined using the same procedure that is used with looking up an item. After the item is found, the node is then removed from the linked list and added to a freed linked list so that the node can be used with future calls on the add procedure.

## Using the *Dictionary(TKey,TValue)* Class

You can refer to the "Uses of Associative Arrays" section in Chapter 2 regarding when to use the *Dictionary(T)* class, but in general, you use the class when you need to access data stored in a collection by using a unique key. Because the *Dictionary(T)* class is the preferred way of creating an associative array, the *Hashtable* class is not discussed in this book. Items in the associative array are represented in the book by using the notation  $[[K_0, V_0], [K_1, V_1], [K_2, V_2], \dots, [K_N, V_N]]$ , where  $[K_0, V_0]$  is the first key value pair in the dictionary,  $[K_1, V_1]$  is the second, and so on.

You need the following method to execute some of the examples.

```
C#
static void Print<T>(ArrayEx<T> list)
{
    Console.Write("[");
    bool first = true;
    for (int i = 0; i < list.Count; ++i)
```

```

    {
        if (!first)
        {
            Console.Write(",");
        }
        first = false;
        Console.Write(list[i]);
    }
    Console.WriteLine("]");
}

```

**Visual Basic**

```

Sub Print(Of T)(ByVal array As T())
    Console.Write("[")
    Dim first As Boolean = True
    For Each item As T In Array
        If (Not first) Then
            Console.Write(",")
        End If
        first = False
        Console.Write(item)
    Next
    Console.WriteLine("]")
End Sub

```

## Creating a Dictionary

You can create a dictionary by using one of the six constructors described in the following sections.

***Dictionary(TKey, TValue)()***

This constructor creates an empty dictionary.

**C#**

```
Dictionary<string, int> dictionary = new Dictionary<string, int>();
```

**Visual Basic**

```
Dim dictionary As New Dictionary(Of String, Integer)()
```

***Dictionary(TKey, TValue)(IDictionary(TKey, TValue) dictionary)***

This constructor creates a new dictionary and copies the items from the *dictionary* parameter over to the new dictionary.

**C#**

```
Dictionary<string, int> dictionary = new Dictionary<string, int>();  
  
dictionary["one"] = 1;  
dictionary["two"] = 2;  
dictionary["three"] = 3;  
  
Dictionary<string, int> str2num = new Dictionary<string, int>(dictionary);
```

**Visual Basic**

```
Dim dictionary As Dictionary(Of String, Integer) = New Dictionary(Of String, Integer)()  
  
dictionary("one") = 1  
dictionary("two") = 2  
dictionary("three") = 3  
  
Dim str2num As New Dictionary(Of String, Integer)(dictionary)
```

The *str2num* variable is created with the items *[[one, 1],[two, 2],[three, 3]]*.

***Dictionary(TKey, TValue)(IEqualityComparer(TKey) comparer)***

This constructor creates a new dictionary with the specified comparer.

**C#**

```
Dictionary<string, int> dictionary = new Dictionary<string, int>  
    (StringComparer.CurrentCultureIgnoreCase);  
  
dictionary["one"] = 1;  
dictionary["two"] = 2;  
dictionary["three"] = 3;  
  
int one = dictionary["One"];
```

**Visual Basic**

```
Dim dictionary As New Dictionary(Of String, Integer) _  
    (StringComparer.CurrentCultureIgnoreCase)  
  
dictionary("one") = 1  
dictionary("two") = 2  
dictionary("three") = 3  
  
Dim one As Integer = dictionary("One")
```

The preceding code creates a new dictionary object with a string comparer that ignores case. If the default comparer is used, it does not ignore case and throws an exception on *int one = dictionary["One"]*; for C# or *Dim one as Integer = dictionary("One")* for Visual Basic.

### ***Dictionary(TKey, TValue)(int capacity)***

This constructor creates a new dictionary object with an initial capacity the same size as the one specified.

#### C#

```
Dictionary<string, int> dictionary = new Dictionary<string, int>(200);
```

#### Visual Basic

```
Dim dictionary As New Dictionary(Of String, Integer)(200)
```

A dictionary is created with a capacity of 200.

### ***Dictionary(TKey, TValue)(IDictionary(TKey, TValue) dictionary, IEqualityComparer(TKey) comparer)***

This constructor creates a new dictionary with the specified comparer and copies the items from the *dictionary* parameter over to the new dictionary.

#### C#

```
Dictionary<string, int> dictionary = new Dictionary<string, int>();

dictionary["one"] = 1;
dictionary["two"] = 2;
dictionary["three"] = 3;

Dictionary<string, int> str2num = new Dictionary<string, int>(dictionary,
    StringComparer.CurrentCultureIgnoreCase);
```

#### Visual Basic

```
Dim dictionary As New Dictionary(Of String, Integer)()
```

```
dictionary("one") = 1
dictionary("two") = 2
dictionary("three") = 3
```

```
Dim str2num New Dictionary(Of String, Integer)(dictionary, _
    StringComparer.CurrentCultureIgnoreCase)
```

The *str2num* variable is created with the items *[[one, 1],[two, 2],[three, 3]]* and the *StringComparer.CurrentCultureIgnoreCase* comparer.

### ***Dictionary(TKey, TValue)(int capacity, IEqualityComparer(TKey) comparer)***

This constructor creates a new dictionary with the specified capacity and comparer.

#### C#

```
Dictionary<string, int> dictionary = new Dictionary<string, int>(25,
    StringComparer.CurrentCultureIgnoreCase);
```

**Visual Basic**

```
Dim dictionary As New Dictionary(Of String, Integer) _  
(25, StringComparer.CurrentCultureIgnoreCase)
```

A dictionary is created with a capacity of 25 and the *StringComparer.CurrentCultureIgnoreCase* comparer.

## Adding Items to a Dictionary

Items can be associated with a key through the *Add* method or *Item* property.

### **void Add(TKey key, TValue value)**

The *Add* method can be used to add a value and key to the dictionary. An *ArgumentException* is thrown if the key already exists in the dictionary. The value can be *null* but not the key. The following example associates people with an employee ID.

**C#**

```
Dictionary<int, String> dictionary = new Dictionary<int, String>();  
  
dictionary.Add(9688, "Barbara Zighetti");  
dictionary.Add(9689, "Eric Gruber");
```

**Visual Basic**

```
Dim dictionary As New Dictionary(Of Integer, String)()  
  
dictionary.Add(9688, "Barbara Zighetti")  
dictionary.Add(9689, "Eric Gruber")
```

The dictionary contains *[[9688, Barbara Zighetti],[9689, Eric Gruber]]* after the code is executed. "Barbara Zighetti" is associated with the employee ID of 9688, and "Eric Gruber" is associated with the employee ID of 9689.

### **TValue Item[TKey key] { set; }**

You can associate a key to a value by using the *Item* property. The advantage of using the *Item* property over the *Add* method is that it overwrites a previous value if the key you specified is already in the dictionary. The value can be *null* but not the key.

**C#**

```
Dictionary<int, String> dictionary = new Dictionary<int, String>();  
  
dictionary[9688] = "Barbara Zighetti";  
dictionary[9689] = "Eric Gruber";
```

**Visual Basic**

```
Dim dictionary As New Dictionary(Of Integer, String)()

dictionary(9688) = "Barbara Zighetti"
dictionary(9689) = "Eric Gruber"
```

The dictionary contains `[[9688, Barbara Zighetti],[9689, Eric Gruber]]` after the code is executed. "Barbara Zighetti" is associated with the employee ID of 9688, and "Eric Gruber" is associated with the employee ID of 9689.

## Removing Items from a Dictionary

Items can be removed from the dictionary by using the *Clear* and *Remove* methods.

***void Clear()***

The *Clear* method removes all keys and values from the dictionary. The capacity of the dictionary does not change.

**C#**

```
Dictionary<int, String> dictionary = new Dictionary<int, String>();

dictionary[9688] = "Barbara Zighetti";
dictionary[9689] = "Eric Gruber";

dictionary.Clear();
```

**Visual Basic**

```
Dim dictionary As New Dictionary(Of Integer, String)()

dictionary(9688) = "Barbara Zighetti"
dictionary(9689) = "Eric Gruber"

dictionary.Clear()
```

The dictionary is changed from `[[9688, Barbara Zighetti],[9689, Eric Gruber]]` to `[]` after the code is executed. The *Clear* method removes all keys and values from the dictionary.

***bool Remove(TKey key)***

The *Remove* method removes the value associated with the specified key. If the key doesn't exist in the dictionary, *false* is returned; otherwise *true* is returned.

**C#**

```
Dictionary<int, String> dictionary = new Dictionary<int, String>();  
  
dictionary[9688] = "Barbara Zighetti";  
dictionary[9689] = "Eric Gruber";  
  
dictionary.Remove(9689);
```

**Visual Basic**

```
Dim dictionary As New Dictionary(Of Integer, String)()  
  
dictionary(9688) = "Barbara Zighetti"  
dictionary(9689) = "Eric Gruber"  
  
dictionary.Remove(9689)
```

The dictionary is changed from *[9688, Barbara Zighetti],[9689, Eric Gruber]* to *[9688, Barbara Zighetti]* after the code is executed. The *Remove* method removes the entry *[9689, Eric Gruber]* because it is associated with the specified key 9689.

## Retrieving Values from the Dictionary by Using a Key

Values can be retrieved from the dictionary through a key by using the *TryGetValue* method and the *Item* property.

### *bool TryGetValue(TKey key, out TValue value)*

The *TryGetValue* method attempts to get the value associated with the specified key by using the *ContainsKey* method and the *Item* property. If the key doesn't exist in the dictionary, it returns *false*; otherwise it returns *true*. Because the method does not throw an exception when the key is not found, it is more efficient to use the *TryGetValue* method than the *Item* property when the key doesn't exist.

**C#**

```
Dictionary<int, String> dictionary = new Dictionary<int, String>();  
  
dictionary[9688] = "Barbara Zighetti";  
dictionary[9689] = "Eric Gruber";  
  
try  
{  
    string name;  
  
    if (dictionary.TryGetValue(9999, out name))  
    {  
        Console.WriteLine("\'{0}\' is associated with 9999", name);  
    }  
}
```

```

        else
    {
        Console.WriteLine("9999 wasn't found in the dictionary");
    }
}
catch (Exception)
{
    Console.WriteLine("An exception was thrown.");
}

```

### Visual Basic

```

Dim dictionary As New Dictionary(Of Integer, String)()

dictionary(9688) = "Barbara Zighetti"
dictionary(9689) = "Eric Gruber"
Try
    Dim name As String = ""

    If (dictionary.TryGetValue(9999, name)) Then
        Console.WriteLine("""{0}"" is associated with 9999", name)
    Else
        Console.WriteLine("9999 wasn't found in the dictionary")
    End If
Catch
    Console.WriteLine("An exception was thrown.")
End Try

```

### Output

9999 wasn't found in the dictionary

As you can see, the *TryGetValue* does not throw an exception if the key doesn't exist in the dictionary. A *false* is returned if the key doesn't exist; otherwise *true* is returned.

### *TValue Item[TKey key] { get; }*

The *Item get* property can be used to retrieve values from the dictionary by using the specified key. A *KeyNotFoundException* exception is thrown if the key isn't located in the dictionary.

#### C#

```

Dictionary<int, String> dictionary = new Dictionary<int, String>();

dictionary[9688] = "Barbara Zighetti";
dictionary[9689] = "Eric Gruber";

Console.WriteLine("\'{0}\' is associated with 9689", dictionary[9689]);

try
{
    Console.WriteLine("\'{0}\' is associated with 9690", dictionary[9690]);
}
catch (KeyNotFoundException ex)
{
    Console.WriteLine("9890 wasn't found in the dictionary");
}

```

### Visual Basic

```
Dim dictionary As New Dictionary(Of Integer, String)()

dictionary(9688) = "Barbara Zighetti"
dictionary(9689) = "Eric Gruber"

Console.WriteLine("'{0}' is associated with 9689", dictionary(9689))

Try
    Console.WriteLine("'{0}' is associated with 9690", dictionary(9690))
Catch ex As KeyNotFoundException
    Console.WriteLine("9890 wasn't found in the dictionary")
End Try
```

### Output

```
"Eric Gruber" is associated with 9689
9890 wasn't found in the dictionary
```

As you can see, the *Item* property returns the value if the key exists, or it throws a *KeyNotFoundException* if it doesn't.

## Checking the Dictionary

You can check the contents of the dictionary without altering it by using the *ContainsKey* and *ContainsValue* methods and the *Count*, *Keys*, and *Values* properties.

### *bool ContainsKey(TKey key)*

The *ContainsKey* method can be used to check whether the specified key is located in the dictionary. *ContainsKey* returns *true* if the key exists in the dictionary; otherwise it returns *false*.

#### C#

```
Dictionary<int, String> dictionary = new Dictionary<int, String>();

dictionary[9688] = "Barbara Zighetti";
dictionary[9689] = "Eric Gruber";

if (dictionary.ContainsKey(9689))
{
    Console.WriteLine("9689 was found in the dictionary");
}
else
{
    Console.WriteLine("9689 was not found in the dictionary");
}

if (dictionary.ContainsKey(9690))
```

```

{
    Console.WriteLine("9690 was found in the dictionary");
}
else
{
    Console.WriteLine("9690 was not found in the dictionary");
}

```

**Visual Basic**

```

Dim dictionary As New Dictionary(Of Integer, String)()

dictionary(9688) = "Barbara Zighetti"
dictionary(9689) = "Eric Gruber"

If (dictionary.ContainsKey(9689)) Then
    Console.WriteLine("9689 was found in the dictionary")
Else
    Console.WriteLine("9689 was not found in the dictionary")
End If

If (dictionary.ContainsKey(9690)) Then
    Console.WriteLine("9690 was found in the dictionary")
Else
    Console.WriteLine("9690 was not found in the dictionary")
End If

```

**Output**

9689 was found in the dictionary  
9690 was not found in the dictionary

The code outputs the results of using the *ContainsKey* method with 9689 and 9690 on a dictionary that contains *[[9688, Barbara Zighetti],[9689, Eric Gruber]]*.

***bool ContainsValue(TValue value)***

The *ContainsValue* method can be used to check whether the specified value is located in the dictionary. *ContainsValue* returns *true* if the value exists in the dictionary; otherwise it returns *false*. The method attempts to find the value by using a linear search with the default equality comparer.

**C#**

```

Dictionary<int, String> dictionary = new Dictionary<int, String>();

dictionary[9688] = "Barbara Zighetti";
dictionary[9689] = "Eric Gruber";

if (dictionary.ContainsValue("Barbara Zighetti"))
{
    Console.WriteLine("\\"Barbara Zighetti\\" was found in the dictionary");
}
else

```

```
{  
    Console.WriteLine("\\"Barbara Zighetti\\" was not found in the dictionary");  
}  
  
if (dictionary.ContainsKey("David Wright"))  
{  
    Console.WriteLine("\\"David Wright\\" was found in the dictionary");  
}  
else  
{  
    Console.WriteLine("\\"David Wright\\" was not found in the dictionary");  
}
```

### Visual Basic

```
Dim dictionary As New Dictionary(Of Integer, String)()  
  
dictionary(9688) = "Barbara Zighetti"  
dictionary(9689) = "Eric Gruber"  
  
If (dictionary.ContainsKey("Barbara Zighetti")) Then  
    Console.WriteLine("""Barbara Zighetti"" was found in the dictionary")  
Else  
    Console.WriteLine("""Barbara Zighetti"" was not found in the dictionary")  
End If  
  
If (dictionary.ContainsKey("David Wright")) Then  
    Console.WriteLine("""David Wright"" was found in the dictionary")  
Else  
    Console.WriteLine("""David Wright"" was not found in the dictionary")  
End If
```

### Output

```
"Barbara Zighetti" was found in the dictionary  
"David Wright" was not found in the dictionary
```

The code outputs the results of using the *ContainsValue* method with *"Barbara Zighetti"* and *"David Wright"* on a dictionary that contains *[[9688, Barbara Zighetti],[9689, Eric Gruber]]*.

### *int Count*

The *Count* property returns the number of key value pairs in the dictionary.

#### C#

```
Dictionary<int, String> dictionary = new Dictionary<int, String>();  
  
dictionary[9688] = "Barbara Zighetti";  
dictionary[9689] = "Eric Gruber";  
  
Console.WriteLine("The dictionary contains {0} items.", dictionary.Count);
```

**Visual Basic**

```
Dim dictionary As New Dictionary(Of Integer, String)()

dictionary(9688) = "Barbara Zighetti"
dictionary(9689) = "Eric Gruber"

Console.WriteLine("The dictionary contains {0} items.", dictionary.Count)
```

**Output**

```
The dictionary contains 2 items.
```

***Dictionary<TKey,TValue>.KeyCollection Keys***

The *Keys* property can be used to retrieve all the keys in the dictionary. The keys are not in a specific order but are returned in the same order as their associated values are returned from the *Values* property. It is important to note that the *Keys* collection is dynamic. If subsequent additions/deletions are made to the dictionary, they are reflected in the *Keys* collection.

**C#**

```
Dictionary<int, String> dictionary = new Dictionary<int, String>();
Dictionary<int, String>.KeyCollection keys = dictionary.Keys;

dictionary[9688] = "Barbara Zighetti";
dictionary[9689] = "Eric Gruber";

Console.WriteLine("The keys in the dictionary are:");
Print(keys.ToArray());

Console.WriteLine("Adding [9687,\\"David Wright\\"]');
dictionary[9687] = "David Wright";

Console.WriteLine("The keys in the dictionary are:");
Print(keys.ToArray());
```

**Visual Basic**

```
Dim dictionary as New Dictionary(Of Integer, String)()
Dim keys As Dictionary(Of Integer, String).KeyCollection = dictionary.Keys

dictionary(9688) = "Barbara Zighetti"
dictionary(9689) = "Eric Gruber"

Console.WriteLine("The keys in the dictionary are:")
Print(keys.ToArray())

Console.WriteLine("Adding [9687,\\"David Wright\\"]")
dictionary(9687) = "David Wright"

Console.WriteLine("The keys in the dictionary are:")
Print(keys.ToArray())
```

## Output

```
The keys in the dictionary are:  
[9688,9689]  
Adding [9687,"David Wright"]  
The keys in the dictionary are:  
[9688,9689,9687]
```

The dictionary contains `[[9688, Barbara Zighetti],[9689, Eric Gruber]]` at the first `Print` but then is changed to `[[9688, Barbara Zighetti],[9689, Eric Gruber],[9687, David Wright]]`. Notice how the collection changes to reflect the update.

## *Dictionary< TKey, TValue >.ValueCollection Values*

The `Values` property can be used to retrieve all the values in the dictionary. The values are not in a specific order but are in the same order as the associated keys in the `Keys` property. As with the `Keys` collection, the `Values` collection is dynamic. If subsequent additions/deletions are made to the dictionary, they are reflected in the `Values` collection.

### C#

```
Dictionary<int, String> dictionary = new Dictionary<int, String>();  
Dictionary<int, String>.ValueCollection values = dictionary.Values;  
  
dictionary[9688] = "Barbara Zighetti";  
dictionary[9689] = "Eric Gruber";  
  
Console.WriteLine("The values in the dictionary are: ");  
Print(values.ToArray());  
  
Console.WriteLine("Adding [9687,\"David Wright\"]");  
dictionary[9687] = "David Wright";  
  
Console.WriteLine("The values in the dictionary are: ");  
Print(values.ToArray());
```

### Visual Basic

```
Dim dictionary As New Dictionary(Of Integer, String)()  
Dim values As Dictionary(Of Integer, String).ValueCollection = dictionary.Values  
  
dictionary(9688) = "Barbara Zighetti"  
dictionary(9689) = "Eric Gruber"  
  
Console.WriteLine("The values in the dictionary are: ")  
Print(values.ToArray())  
  
Console.WriteLine("Adding [9687,\"David Wright\"]")  
dictionary(9687) = "David Wright"  
  
Console.WriteLine("The values in the dictionary are: ")  
Print(values.ToArray())
```

### Output

```
The values in the collection are:  
[Barbara Zighetti, Eric Gruber]  
Adding [9687, "David Wright"]  
The values in the collection are:  
[Barbara Zighetti, Eric Gruber, David Wright]
```

The dictionary contains `[[9688, Barbara Zighetti],[9689, Eric Gruber]]` at the first `Print` but then is changed to `[[9688, Barbara Zighetti],[9689, Eric Gruber],[9687, David Wright]]`. Notice how the collection changes to reflect the update.

## BitArray Overview

You can use the `BitArray` class to manipulate an array of Booleans that represent bits. Each bit value is represented as a Boolean.

The `BitArray` class stores each bit in a compact form. Bit 0 could represent the most significant bit (MSB) or the least significant bit (LSB). This book displays the 0 bit as the LSB and displays the bits in groups of four. The value `0000 0010` would represent only bit 1 being set in a group of eight bits, `0001 0010` would represent only bits 1 and 4 being set in a group of eight bits. Displaying bits this way helps you if you need to put the bits in a calculator to get the integer form.

The `BitArray` class is defined in the `System.Collections` namespace and has no generic equivalent.

## Using the `BitArray` Class

You can use the `BitArray` class when you need to perform bit manipulations. One example would be if you were dealing with flags. The `BitArray` class makes it easier to understand what you are doing for other developers. However, it doesn't outperform the bitwise operators.

### Creating a `BitArray`

You can create a `BitArray` object using the six constructors. The `BitArray` class does not have a default constructor.

#### `BitArray(BitArray bits)`

Use this constructor when you want to create a copy of another `BitArray` instance.

**C#**

```
BitArray x = new BitArray(8, false);
x.Set(2, true);

BitArray y = new BitArray(x);
```

After the code executes, *x* and *y* both contain eight bits, and bit 2 will be the only bit set *true*.

### ***BitArray(Boolean[] values)***

This constructor creates a *BitArray* instance with the bits set according to the values passed into it. The first element is bit 0, the second is bit 1, and so on.

**C#**

```
BitArray bits = new BitArray(new bool[] { false, true, false, false });
```

**Visual Basic**

```
Dim bits As New BitArray(New Boolean() {False, True, False, False})
```

The executed code creates the bit array as 0010.

### ***BitArray(Byte[] bytes)***

This constructor creates a *BitArray* by using the specified bytes. Each byte represents a set of eight bits. The first byte represents the first set, the second byte the second set, and so on. The first set would be bits 0 through 7, the second set would be the next eight (bits 8 through 15), the third set would be the next eight bits (16 through 23), and so on. The LSB bit of each byte is copied first; *bytes[0] & 1* would be bit 0, *bytes[0] & 2* would be bit 1, *bytes[1] & 0* would be bit 8, and so on.

**C#**

```
BitArray bits = new BitArray(new byte[] { 1, 2 });
```

**Visual Basic**

```
Dim bits As New BitArray(New Byte() {1, 2})
```

The executed code creates a *BitArray* that is set to 0000 0010 0000 0001. Byte 0 would set bits 0 through 7 to 0000 0001, and byte 1 would set bits 8 through 15 to 0000 0010. This book then shows the bits as 15 through 0 by flipping the bits to 0000 0010 0000 0001.

### ***BitArray(Int32 length, Boolean defaultValue)***

This constructor creates a *BitArray* with the number of bits specified in *length*. Each bit is set to the value in *defaultValue*.

**C#**

```
BitArray bits = new BitArray(4, true);
```

**Visual Basic**

```
Dim bits As New BitArray(4, True)
```

The code creates a *BitArray* initialized as *1111*.

### ***BitArray(Int32 length)***

Calling this constructor is the same as calling the previous constructor with *defaultValue* set to *false*. See the *BitArray(int32 length, Boolean defaultValue)* constructor discussed in the previous section for more details.

### ***BitArray(Int32[] values)***

This constructor works the same way as the *BitArray(Byte[])* constructor except each set now represents 32 bits instead of 8. So the first set would be *0* through *31*, the second set would be *32* through *62*, and so on. See the *BitArray(Bytes[] bytes)* constructor described earlier in this chapter for more details.

## **Accessing Bits in the *BitArray* Class**

Individual bits can be accessed using the *Set*, *SetAll*, and *Get* methods as well as the *Item* property.

### ***void Set(int index, bool value)***

You can use the *Set* method to set an individual bit in the array.

**C#**

```
BitArray bits = new BitArray(4);

bits.Set(0, true);
bits.Set(1, true);
```

**Visual Basic**

```
Dim bits As New BitArray(4)

bits.Set(0, True)
bits.Set(1, True)
```

The preceding code creates a four-bit array and changes the *bits* from the initial value of *0000* to *0011*.

### ***void SetAll(bool value)***

The *SetAll* method lets you set all the bits to the specified value.

#### C#

```
BitArray bits = new BitArray(4);  
  
bits.SetAll(true);
```

#### Visual Basic

```
Dim bits As New BitArray(4)  
  
bits.SetAll(True)
```

The preceding code creates a four-bit array and changes the *bits* from an initial value of *0000* to *1111*.

### ***bool Get(int index)***

The *Get* method allows you to access a specific bit in the array.

#### C#

```
BitArray bits = new BitArray(new byte[] { 3 });  
  
Console.Write("3 is stored as ");  
  
for (int i = bits.Length - 1; i >= 0; --i)  
{  
    Console.Write(bits.Get(i) ? "1" : "0");  
}  
Console.WriteLine();
```

#### Visual Basic

```
Dim bits As New BitArray(New Byte() {3})  
  
Console.Write("3 is stored as ")  
  
For i As Integer = bits.Length - 1 To 0 Step -1  
    Console.Write>If(bits.Get(i), "1", "0"))  
Next  
Console.WriteLine()
```

#### Output

```
3 is stored as 00000011
```

The executed code displays the 3 as *0000 0011*. You also get the number 3 if you enter *0000 0011* into a calculator in binary form and convert it to decimal.

### ***bool Item[int index] {get; set; }***

The *Index* property is used to set or get a bit at the specified index.

**C#**

```
BitArray bits = new BitArray(4);

Console.WriteLine("bit 0 is {0}", bits[0]);
Console.WriteLine("Negating bit 0");
bits[0] = !bits[0];
Console.WriteLine("bit 0 is now {0}", bits[0]);
```

**Visual Basic**

```
Dim bits As New BitArray(4)

Console.WriteLine("bit 0 is {0}", bits(0))
Console.WriteLine("Negating bit 0")
bits(0) = Not bits(0)
Console.WriteLine("bit 0 is now {0}", bits(0))
```

**Output**

```
bit 0 is False
Negating bit 0
bit 0 is now True
```

The executed code negates the value of bit 0.

## Using the *BitArray* Class for Bit Operations

You can use the *And*, *Not*, *Or*, and *Xor* methods to manipulate the bits of the *BitArray* class. One common reason for doing this is to treat the array values as flags.

Each bit could represent a flag, as shown in the following *enum*.

**C#**

```
[Flags()]
public enum FileFlags
{
    IsArchived = 0,
    IsReadOnly = 1,
    IsCompressed = 2,
    IsHidden = 3,
    NUM_FLAGS
}
```

**Visual Basic**

```
<Flags()>_
Public Enum FileFlags
    IsArchived = 0
    IsReadOnly = 1
    IsCompressed = 2
    IsHidden = 3
    NUM_FLAGS
End Enum
```

## Using Bitwise Operations on Flags

Let's say that you have a number of flags for your application that you want to store in compact form. The flags are called  $F_0, F_1, \dots, F_N$ . To store these flags, you first need a data type whose bit count is greater than or equal to the number of flags you need. You could also get by with using more than one data type to hold the flags, but keep it simple for now. So, if you have 18 flags, you could store them in a 32-bit integer. To store the flags in the data type, each flag number will represent the bit it is stored in. So,  $F_X$  means that it occupies bit  $X$ , which in decimal form would be  $2^X$ . To store the value, you would say  $value = 2^X = (1 << X)$  (using C#).

But what if you want to store more than one flag in the same variable? This is when the *OR* operation comes in. Using *OR*, you can combine more than one flag because flags are represented as bits and the *OR* operation sets the resulting bit if either one of the operands have the bit set. So, if you wanted to store  $F_i$  and  $F_j$ , you would say  $int value = 2^i | 2^j$  (using C#).

Now how could you check to see whether the *bit*  $F_X$  is set? This is where the *AND* operation comes in. If you *AND* the *value* with  $F_X$ , the result would be *0* if the bit  $F_X$  is not set; the result would be  $F_X$  if it is set. The *AND* operation sets the bit in the result if and only if both operands have the same bit set. So to check and see whether  $F_X$  is contained in *value*, you could do  $(value & 2^X) == 2^X$  (using C#).

Now what if you wanted to remove a flag from the value? This is where the *NOT* operation comes in. The *NOT* operation turns off all bits that are on and turns on all bits that are off. So,  $\sim(F_X)$  would mean all flags except  $F_X$  are on. If you perform the operation  $(\sim(F_X) & value)$ , you will be *ANDing* every flag except  $F_X$  with *1*. This results in any flag that was on staying on and  $F_X$  being turned off.

Each value represents the file being archived, read-only, compressed, or hidden. The first value is for bit *0*, the second is for bit *1*, and so on. You could have a *BitArray* represent those flags. To display the flags, you could define the following method.

```
C#
static string FlagsToString(BitArray bits)
{
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < (int)FileFlags.NUM_FLAGS; ++i)
    {
        if (bits[i])
        {
            if (sb.Length > 0)
            {
                sb.Append(" | ");
            }
            sb.Append(i);
        }
    }
}
```

```

        sb.Append(((FileFlags)i).ToString());
    }

    return sb.ToString();
}

```

### Visual Basic

```

Function FlagsToString(ByVal bits As BitArray) As String
    Dim sb As New System.Text.StringBuilder()

    For i As Integer = 0 To FileFlags.NUM_FLAGS - 1
        If (bits(i)) Then
            If (sb.Length > 0) Then
                sb.Append(" | ")
            End If
            sb.Append(DirectCast(i, FileFlags).ToString())
        End If
    Next

    Return sb.ToString()
End Function

```

This method returns a string that contains all the flags that are true in the *BitArray* by walking each flag and checking to see whether it is set in the *BitArray*.

To create a *BitArray* from a group of flags, you could do as follows.

### C#

```

static BitArray FlagsToBitArray(params FileFlags[] flags)
{
    BitArray retval = new BitArray((int)FileFlags.NUM_FLAGS);

    foreach (FileFlags flag in flags)
    {
        retval.Set((int)flag, true);
    }

    return retval;
}

```

### Visual Basic

```

Function FlagsToBitArray(ByVal ParamArray flags As FileFlags()) As BitArray
    Dim retval As New BitArray(Int(FileFlags.NUM_FLAGS))

    For Each flag As FileFlags In flags
        retval.Set(Int(flag), True)
    Next

    Return retval
End Function

```

The next sections use these methods to demonstrate the following *BitArray* methods.

## BitArray And(BitArray value)

The *And* method performs a bitwise *AND* on all the bits of the current *BitArray* and the specified *BitArray*. The method creates another *BitArray* object that contains the result of the *AND* on each bit. If the same indexed bit of the specified *BitArray* and the current *BitArray* are *true*, *true* is set in the returned index for that index bit; otherwise *false* is set.

### C#

```
BitArray bits = FlagsToBitArray(FileFlags.IsCompressed, FileFlags.IsArchived,
                               FileFlags.isHidden);
Console.WriteLine("bits = {0}", FlagsToString(bits));
Console.WriteLine("Keep only IsArchived on if it is on");
bits = bits.And(FlagsToBitArray(FileFlags.IsArchived));
Console.WriteLine("bits = {0}", FlagsToString(bits));
```

### Visual Basic

```
Dim bits As BitArray = FlagsToBitArray(FileFlags.IsCompressed, FileFlags.IsArchived, _
                                       FileFlags.isHidden)
Console.WriteLine("bits = {0}", FlagsToString(bits))
Console.WriteLine("Keep only IsArchived on if it is on")
bits = bits.And(FlagsToBitArray(FileFlags.IsArchived))
Console.WriteLine("bits = {0}", FlagsToString(bits))
```

### Output

```
bits = IsArchived | IsCompressed | IsHidden
Keep only IsArchived on if it is on
bits = IsArchived
```

## BitArray Not()

The *Not* method performs a bitwise *NOT* on all the bits of the current *BitArray*. The method creates another *BitArray* object that contains the result of the *NOT* on each bit. The method essentially flips each bit of the current *BitArray*.

### C#

```
BitArray bits = FlagsToBitArray(FileFlags.IsCompressed, FileFlags.IsArchived,
                               FileFlags.isHidden);
Console.WriteLine("bits = {0}", FlagsToString(bits));
Console.WriteLine("Flip all flags");
bits = bits.Not();
Console.WriteLine("bits = {0}", FlagsToString(bits));
```

### Visual Basic

```
Dim bits As BitArray = FlagsToBitArray(FileFlags.IsCompressed, _
                                       FileFlags.IsArchived, FileFlags.isHidden)
Console.WriteLine("bits = {0}", FlagsToString(bits))
Console.WriteLine("Flip all flags")
bits = bits.Not()
Console.WriteLine("bits = {0}", FlagsToString(bits))
```

### Output

```
bits = IsArchived | IsCompressed | IsHidden
Flip all flags
bits = IsReadOnly
```

### *BitArray Or(BitArray value)*

The *Or* method performs a bitwise *OR* on all the bits of the current *BitArray* and the specified *BitArray*. The method creates another *BitArray* object that contains the result of the *OR* on each bit. If either the bit of the specified *BitArray* or the corresponding bit in the current *BitArray* are *true*, *true* is returned for that bit; otherwise *false* is returned.

#### C#

```
BitArray bits = FlagsToBitArray(FileFlags.IsCompressed, FileFlags.IsArchived,
                               FileFlags.isHidden);
Console.WriteLine("bits = {0}", FlagsToString(bits));
Console.WriteLine("Turning IsReadOnly flag on");
bits = bits.Or(FlagsToBitArray(FileFlags.IsReadOnly));
Console.WriteLine("bits = {0}", FlagsToString(bits));
```

#### Visual Basic

```
Dim bits As BitArray = FlagsToBitArray(FileFlags.IsCompressed, _
                                       FileFlags.IsArchived, FileFlags.isHidden)
Console.WriteLine("bits = {0}", FlagsToString(bits))
Console.WriteLine("Turning IsReadOnly flag on")
bits = bits.Or(FlagsToBitArray(FileFlags.IsReadOnly))
Console.WriteLine("bits = {0}", FlagsToString(bits))
```

### Output

```
bits = IsArchived | IsCompressed | IsHidden
Turning IsReadOnly flag on
bits = IsArchived | IsReadOnly | IsCompressed | IsHidden
```

You may have noticed that this code is like executing *bits[(int)FileFlags.IsReadOnly] = true*.

### *BitArray Xor(BitArray value)*

The *Xor* method performs an exclusive *OR* on all the bits of the current *BitArray* and the specified *BitArray*. The method creates another *BitArray* object that contains the result of the exclusive *OR* on each bit. If either the bit of the specified *BitArray* or the corresponding bit in the current *BitArray* have the same value, *false* is set; otherwise if only one bit is true, *true* is set.

You need the following method to run the code.

**C#**

```
static string PrintString(BitArray bits)
{
    string str = "";
    byte value = 0;

    for (int i = bits.Length - 1; i >= 0; --i)
    {
        if (bits[i])
        {
            value |= (byte)(1 << (i % 8));
        }

        if (i % 8 == 7)
        {
            str = (char)value + str;
            value = 0;
        }
    }

    str = (char)value + str;

    return str;
}
```

**Visual Basic**

```
Function PrintString(ByVal bits As BitArray) As String
    Dim str As String = ""

    Dim value As Byte = 0

    For i As Integer = bits.Length - 1 To 0 Step -1
        If (bits(i)) Then
            value = value Or CByte(1 << (i Mod 8))
        End If

        If ((i Mod 8) = 7) Then
            str = Chr(value) & str
            value = 0
        End If
    Next

    str = Chr(value) + str

    Return str
End Function
```

The method converts a bit array into a string.

A long time ago developers used an *XOR* operation to encrypt text. The following code does just that. I would never suggest using it or showing it to your boss; your boss may wonder why he has spent so much money on security when he could have just used *XOR*. However, you and I realize it can easily be broken.

#### C#

```
BitArray bits = new BitArray(new byte[]
    { (byte)'h', (byte)'e', (byte)'l', (byte)'l', (byte)'o' });
BitArray key = new BitArray(new byte[] { 99, 88, 55, 22, 11 });
Console.WriteLine("string = {0}", PrintString(bits));
Console.WriteLine("Encrypting");
bits = bits.Xor(key);
Console.WriteLine("string = {0}", PrintString(bits));
Console.WriteLine("Decrypting");
bits = bits.Xor(key);
Console.WriteLine("string = {0}", PrintString(bits));
```

#### Visual Basic

```
Dim bits As New BitArray(New Byte() _
    {Asc("h"c), Asc("e"c), Asc("l"c), Asc("l"c), Asc("o"c)})
Dim key As BitArray = New BitArray(New Byte() {99, 88, 55, 22, 11})
Console.WriteLine("string = {0}", PrintString(bits))
Console.WriteLine("Encrypting")
bits = bits.Xor(key)
Console.WriteLine("string = {0}", PrintString(bits))
Console.WriteLine("Decrypting")
bits = bits.Xor(key)
Console.WriteLine("string = {0}", PrintString(bits))
```

#### Output

```
string = hello
Encrypting
string = o=[zd
Decrypting
string = hello
```

## CollectionBase and DictionaryBase Overview

The *CollectionBase* and *DictionaryBase* classes were created before Microsoft .NET generics were invented to give developers an easy way of creating strongly typed collections and dictionaries respectively. Since the addition of generics, there is really no need to use these classes any more. If you want to create a collection of *ints*, it's easier to use the *List<int>* class instead of creating your own class and deriving it from *CollectionBase*. If you need a dictionary that uses strings as keys and integers as values, create a *Dictionary<string,int>* instead of deriving a class from *DictionaryBase*. This section will discuss them briefly in case you find some legacy code that uses them or you find a reason to still use them.

## Using *CollectionBase*

*CollectionBase* is a dynamic array that uses an *ArrayList* as its internal data storage. The *CollectionBase* class exposes methods that allow you to do additional processing before and after *CollectionBase* operates on the internal *ArrayList*. The methods that allow you to do preprocessing begin with *On*, and the ones that allow you to do additional post-processing begin with *On* and end with *Complete*.

Sometimes you may need to bypass the additional processing you created. Two internal protected properties allow you to do so: *InnerList* and *List*. You need to know the difference between these before you move on to the next section.

The *List* property interacts with the internal *ArrayList* by calling the appropriate *On\** method discussed later in this section. The *InnerList* property returns the internal *ArrayList*, which allows you to do operations without having the *On\** methods called.

### **void OnValidate(object value)**

With the *OnValidate* method, you can do custom validation on the object before it is removed, or added to the internal *ArrayList*. This is your chance to check whether the object is a certain type before being added to your type-safe collection. You could also check other business rules, such as whether it is *null*, or whether the object is a bad debt bill before adding it to the bad debt bill collection.

The following code snippets can be used within *OnValidate* to check whether the item is a *Timer*.

#### C#

```
if (value != null && !(value is System.Timers.Timer))
{
    throw new ArgumentException("value must be of type System.Timers.Timer.", "value");
}
```

#### Visual Basic

```
If (Not value Is Nothing And Not (TypeOf value Is System.Timers.Timer)) Then
    Throw New ArgumentException("value must be of type System.Timers.Timer.", "value")
End If
```

The following code snippets can be used within *OnValidate* to check whether the object is the value type *int*.

#### C#

```
if (value == null || !(value is int))
{
    throw new ArgumentException("value must be of type int.", "value");
}
```

**Visual Basic**

```
If (value Is Nothing Or Not (TypeOf value Is Integer)) Then  
    Throw New ArgumentException("value must be of type int.", "value")  
End If
```

**void OnClear() and void OnClearComplete()**

You can use the *OnClear* and *OnClearComplete* methods to do additional processing before and after the *Clear* method call. The framework does the following when the *CollectionBase.Clear* is called.

**C#**

```
OnClear();  
InnerList.Clear();  
OnClearComplete();
```

**Visual Basic**

```
OnClear()  
InnerList.Clear()  
OnClearComplete()
```

In the past, I have found it sometimes useful to store the objects being cleared during the *OnClearComplete* called. Maybe you fire an event after the *Clear* has been done and the object that received the event needs to know what objects have been cleared so that it may remove those items as well. A simple way to accomplish this is to have a property that stores the removed items, such as the following.

**C#**

```
object[] m_clearItems;  
public object[] ClearedItems { get { return m_clearItems; } }
```

**Visual Basic**

```
Dim m_clearItems() As Object  
Public ReadOnly Property ClearedItems() As Object  
    Get  
        Return m_clearItems  
    End Get  
End Property
```

You could then set the cleared items in the array by doing the following in your *OnClear* method.

**C#**

```
m_clearItems = InnerList.ToArray();
```

**Visual Basic**

```
m_clearItems = InnerList.ToArray()
```

Next, you will want to release the memory for the objects after you fire the event in your *OnClearComplete* call, as follows.

**C#**

```
m_clearItems = null;
```

**Visual Basic**

```
m_clearItems = Nothing
```

This implementation has several problems. First, the receiver of the event has to cast the *CollectionBase* to your implementation to interact with the *ClearedItems* property. Second, if the *InnerList.Clear* fails, you have no way of removing the items in *m\_clearItems*. Last but not least, it isn't very thread safe because multiple threads could make simultaneous calls to the *OnClearComplete* on the same object instance, which could cause *m\_clearItems* to be overwritten before you handle it in the initial event. You learn more about threads and how to handle this issue in Chapter 8, "Using Threads with Collections."

***void OnInsert(int index, Object value) and void OnInsertComplete(int index, Object value)***

The *OnInsert* and *OnInsertComplete* methods both have as the argument the object that is being inserted and the index of where the object is being inserted. The following is the code for the *CollectionBase.Insert* method.

**C#**

```
OnValidate(value);
OnInsert(index, value);
InnerList.Insert(index, value);
try
{
    OnInsertComplete(index, value);
}
catch
{
    InnerList.RemoveAt(index);
    throw;
}
```

**Visual Basic**

```
OnValidate(value)
OnInsert(index, value)
InnerList.Insert(index, value)
Try
    OnInsertComplete(index, value)
Catch
    InnerList.RemoveAt(index)
    Throw
End Try
```

The following is the code for the *CollectionBase.Add*.

#### C#

```
OnValidate(value);
OnInsert(InnerList.Count, value);
int index = InnerList.Add(value);
try
{
    OnInsertComplete(index, value);
}
catch
{
    InnerList.RemoveAt(index);
    throw;
}
return index;
```

#### Visual Basic

```
OnValidate(value)
OnInsert(InnerList.Count, value)
Dim index As Integer = InnerList.Add(value)
Try
    OnInsertComplete(index, value)
Catch
    InnerList.RemoveAt(index)
    Throw
End Try
Return index
```

Both methods do the same thing, except the *Add* method must first calculate the index of the item being added before calling the *Add* method. For an *Add*, the index is always *Count* because the item is added to the end of the list. As seen from the preceding code, the *OnInsert* method is called before the *Insert* or *Add* method is called on the internal *ArrayList*. After the item has been successfully added or inserted into the list, the *OnInsertComplete* is called. If you throw an exception in *OnInsertComplete*, the framework automatically removes the added object from the internal *ArrayList* by using the position it was added at. You are responsible for undoing anything you have done in *OnInsert* and *OnInsertComplete* before the exception is thrown. You could do something like the following.

#### C#

```
protected override void OnInsertComplete(int index, object value)
{
    try
    {
        base.OnInsertComplete(index, value);

        // Code to do after the insert has completed
    }
    catch
```

```
{  
    // Undo OnInsert  
  
    throw;  
}  
}
```

### Visual Basic

```
Protected Overrides Sub OnInsertComplete(ByVal index As Integer, ByVal value As Object)  
    Try  
        MyBase.OnInsertComplete(index, value)  
        ' Code to do after the insert has completed  
    Catch  
        ' Undo OnInsert  
        Throw  
    End Try  
End Sub
```

## ***void OnRemove(int index, object value) and void OnRemoveComplete(int index, object value)***

The *OnRemove* and *OnRemoveComplete* methods both have as the argument the object that is being removed and the index that is being removed. The following is the code for the *CollectionBase.Remove* method.

### C#

```
OnValidate(value);  
int index = InnerList.IndexOf(value);  
if (index < 0)  
{  
    throw new ArgumentException("Index out of range");  
}  
OnRemove(index, value);  
InnerList.RemoveAt(index);  
try  
{  
    OnRemoveComplete(index, value);  
}  
catch  
{  
    InnerList.Insert(index, value);  
    throw;  
}
```

### Visual Basic

```
OnValidate(value)  
Dim index As Integer = InnerList.IndexOf(value)  
If (index < 0) Then  
    Throw New ArgumentException("Index out of range")
```

```

End If
OnRemove(index, value)
InnerList.RemoveAt(index)
Try
    OnRemoveComplete(index, value)
Catch
    InnerList.Insert(index, value)
    Throw
End Try

```

As seen in the preceding code, the *OnRemove* method is called before the *RemoveAt* on the internal *ArrayList*. A *RemoveAt* is performed instead of a *Remove* to insure the index and value passed to you are the ones being removed and not another instance of the value that is equal to the one passed in. After the item has been successfully removed, the *OnRemoveComplete* is called. If you throw an exception in *OnRemoveComplete*, the framework automatically adds the removed object back to the internal *ArrayList* at the position it was removed from. You are responsible for undoing anything you have done in *OnRemove* in the *OnRemoveComplete* before the exception is thrown. You could do something like the following.

#### C#

```

protected override void OnRemoveComplete(int index, object value)
{
    try
    {
        base.OnRemoveComplete(index, value);

        // Code to do after the remove has completed
    }
    catch
    {
        // Undo OnRemove

        throw;
    }
}

```

#### Visual Basic

```

Protected Overrides Sub OnRemoveComplete(ByVal index As Integer, ByVal value As Object)
    Try
        MyBase.OnRemoveComplete(index, value)
        ' Code to do after the remove has completed
    Catch
        ' Undo OnRemove
        Throw
    End Try
End Sub

```

**void OnSet(int index, object oldValue, object newValue) and void OnSetComplete(int index, object oldValue, object newValue)**

The *OnSet* and *OnSetComplete* methods are called when the user calls the *set* method of the *Item* property. Both have as the argument the object that is set, the value that the *set* method is overwriting, and the index being set.

**C#**

```
OnValidate(value);
object oldValue = InnerList[index];
OnSet(index, oldValue, value);
InnerList[index] = value;
try
{
    OnSetComplete(index, oldValue, value);
}
catch
{
    InnerList[index] = oldValue;
    throw;
}
```

**Visual Basic**

```
OnValidate(value)
Dim oldValue As Object = InnerList.Item(index)
OnSet(index, oldValue, value)
InnerList.Item(index) = value
Try
    OnSetComplete(index, oldValue, value)
Catch
    InnerList.Item(index) = oldValue
    Throw
End Try
```

As seen in the preceding code, the *OnSet* method is called before the *Item { set; }* is called on the internal *ArrayList*. After the element is successfully set, *OnSetComplete* is called. If you throw an exception in *OnSetComplete*, the framework automatically sets the element at the specified index back to the original value. You are responsible for undoing anything you have done in *OnSet* in the *OnSetComplete* method before the exception is thrown. You could do something like the following.

**C#**

```
protected override void OnSetComplete(int index, object oldValue, object newValue)
{
    try
    {
        base.OnSetComplete(index, oldValue, newValue);

        // Code to do after the set has completed
    }
    catch
```

```

    {
        // Undo OnSet

        throw;
    }
}

```

### Visual Basic

```

Protected Overrides Sub OnSetComplete(ByVal index As Integer, ByVal oldValue As Object, _
                                    ByVal newValue As Object)
    Try
        MyBase.OnSetComplete(index, oldValue, newValue)

        ' Code to do after the set has completed
    Catch
        ' Undo OnSet

        Throw
    End Try
End Sub

```

## Using *DictionaryBase*

*DictionaryBase* is an associative array that uses a *Hashtable* as its internal data storage. The methods in *DictionaryBase* are designed so that you can do additional processing before and after *DictionaryBase* operates on the internal *Hashtable*. To do this, *DictionaryBase* defines methods that allow you to do additional processing before and after operating on the internal *Hashtable*. The methods that allow you to preprocess begin with *On*, and the ones that allow you to do additional post-processing begin with *On* and end with *Complete*.

Sometimes you may need to bypass the additional processing you created. Two internal protected properties allow you to do so: *InnerHashtable* and *Dictionary*. You need to know the difference between these before you move on to the next section.

The *Dictionary* property interacts with the internal *Hashtable* by calling the appropriate *On\** method discussed later in this section. The *InnerHashtable* property returns the internal *Hashtable*, which allows you to do operations without having the *On\** methods called.

### **void *OnValidate(object key, object value)***

With the *OnValidate* method, you can custom validate the value and key before it is removed, retrieved, or added to the internal *Hashtable*. This is your chance to perform type-safe checking on the key and value. Refer to the *OnValidation* section for the *CollectionBase* class regarding how to do type-safe checking.

### **void OnClear() and void OnClearComplete()**

The *OnClear* and *OnClearComplete* methods allow you to do additional processing before and after the *Clear* method call. The framework does the following when the *DictionaryBase.Clear* is called.

#### C#

```
OnClear();
InnerHashtable.Clear();
OnClearComplete();
```

#### Visual Basic

```
OnClear()
InnerHashtable.Clear()
OnClearComplete()
```

In the past, I have found it sometimes useful to store the objects being cleared during the *OnClearComplete* call. You can look at the *OnClear* method in the section titled, “*void OnClear() and void OnClearComplete()*” to get an idea about how to accomplish this.

### **void OnGet()**

The *OnGet* method is called in the *Item { get; }*. The method is implemented as follows.

#### C#

```
object currentValue = InnerHashtable[key];
OnGet(key, currentValue);
return currentValue;

Dim currentValue as Object = InnerHashtable(key)
OnGet(key, currentValue)
return currentValue
```

The *OnGet* method allows you to interact with the object before it is returned to you.

### **void OnInsert(Object key, Object value) and void OnInsertComplete(Object key, Object value)**

The *OnInsert* and *OnInsertComplete* methods both have as the argument the object that is being added and the key to associate with that value. The following is the code for the *DictionaryBase.Add* method.

#### C#

```
OnValidate(key, value);
OnInsert(key, value);
InnerHashtable.Add(key, value);
try
```

```

    {
        OnInsertComplete(key, value);
    }
    catch
    {
        InnerHashtable.Remove(key);
        throw;
    }
}

```

### Visual Basic

```

OnValidate(key, value)
OnInsert(key, value)
InnerHashtable.Add(key, value)
Try
    OnInsertComplete(key, value)
Catch
    InnerHashtable.Remove(key)
    Throw
End Try

```

As seen in the preceding code, the *OnInsert* method is called before the *Add* method is called on the internal *Hashtable*. After the item has been successfully added to the internal *Hashtable*, the *OnInsertComplete* is called. If you throw an exception in *OnInsertComplete*, the framework automatically removes the added object from the internal *Hashtable*. You are responsible for undoing anything you have done in the *OnInsert* and *OnInsertComplete* methods before the exception is thrown. See the section titled “*void OnInsert(int index, Object value)* and *void OnInsertComplete(int index, Object value)*” if you need information about how to handle an exception being thrown in *OnInsertComplete*.

### ***void OnRemove(Object key, Object value) and void OnRemoveComplete(Object key, Object value)***

The *OnRemove* and *OnRemoveComplete* methods both have as the argument the object that is being removed and the key for the object that is being removed. The following is the code for the *DictionaryBase.Remove* method.

```

C#
if (InnerHashtable.Contains(key))
{
    object obj2 = InnerHashtable[key];
    OnValidate(key, obj2);
    OnRemove(key, obj2);
    InnerHashtable.Remove(key);
    try
    {
        OnRemoveComplete(key, obj2);
    }
    catch
}

```

```
{  
    InnerHashtable.Add(key, obj2);  
    throw;  
}  
}
```

### Visual Basic

```
If (InnerHashtable.Contains(key)) Then  
    Dim obj2 As Object = InnerHashtable(key)  
    OnValidate(key, obj2)  
    OnRemove(key, obj2)  
    InnerHashtable.Remove(key)  
    Try  
        OnRemoveComplete(key, obj2)  
    Catch  
        InnerHashtable.Add(key, obj2)  
        Throw  
    End Try  
End If
```

As seen in the preceding code, the *Contains* method of the *InnerHashtable* is first called to verify that the key exists in the dictionary. Then the key's value is retrieved from the internal *Hashtable*, and the *OnValidate* and *OnRemove* methods are called before the *Remove* method is invoked on the internal *Hashtable*. Next, the object is retrieved from the *InnerHashtable* and *OnValidate* is called. The item is added back to the internal *Hashtable* if an exception is encountered in *OnRemoveComplete*. See the section titled "void *OnRemove(int index, object value)* and void *OnRemoveComplete(int index, object value)*" if you need information about how to handle an exception being thrown in *OnRemoveComplete*.

### **void *OnSet(Object key, object oldValue, object newValue)* and void *OnSetComplete(Object key, object oldValue, object newValue)***

The *OnSet* and *OnSetComplete* methods are called when the user calls the *set* method on the *Item* property. Both have as the argument the object that is set, the value that the set is overwriting, and the key of the value being set.

### C#

```
OnValidate(key, value);  
bool flag = true;  
object oldValue = InnerHashtable[key];  
if (oldValue == null)  
{  
    flag = InnerHashtable.Contains(key);  
}  
OnSet(key, oldValue, value);  
InnerHashtable[key] = value;  
try
```

```

    {
        OnSetComplete(key, oldValue, value);
    }
    catch
    {
        if (flag)
        {
            InnerHashtable[key] = oldValue;
        }
        else
        {
            InnerHashtable.Remove(key);
        }
        throw;
    }
}

```

### Visual Basic

```

OnValidate(key, value)
Dim flag As Boolean = True
Dim oldValue As Object = InnerHashtable.Item(key)
If (oldValue Is Nothing) Then
    flag = InnerHashtable.Contains(key)
End If
OnSet(key, oldValue, value)
InnerHashtable.Item(key) = value
Try
    OnSetComplete(key, oldValue, value)
Catch
    If flag Then
        InnerHashtable.Item(key) = oldValue
    Else
        InnerHashtable.Remove(key)
    End If
    Throw
End Try

```

As seen in the preceding code, the *OnSet* method is called before the *Item { set; }* is called on the internal *Hashtable*. Before the call is made, the code attempts to get the original value from the internal *Hashtable*. If it retrieves a *null*, it checks to see whether the internal *Hashtable* actually contains the key so that it may reset it if the *OnSetComplete* throws an exception. After the element has been successfully set, the *OnSetComplete* is called. If an error is encountered in *OnSetComplete*, the code sets the key back to its original value. Please refer to the section titled, "void *OnSet(int index, object oldValue, object newValue)* and void *OnSetComplete(int index, object oldValue, object newValue)*" if you need more information about how to handle an exception being thrown in *OnSetComplete*.

## HashSet(*T*) Overview

A *set* is a type of collection that cannot contain duplicates. For example, [1, 3, 1, 4] is not a set because 1 is in element 0 and 2.

Because the *HashSet(*T*)* has a hash-based implementation, the *Add*, *Remove*, and *Contains* methods are  $O(1)$  operations instead of  $O(n)$  like *List(*T*).Remove* and *List(*T*).Contains*.

## Using the *HashSet(*T*)* Class

You should consider using a *HashSet(*T*)* if you need to ensure that your list doesn't contain duplicates and/or you need to do set operations such as union, intersection, and symmetric difference.

### Creating a *HashSet(*T*)*

You can create a *HashSet(*T*)* in four ways.

#### *HashSet(*T*) ()*

You can create an empty *HashSet(*T*)* instance by writing the following.

##### C#

```
HashSet<int> a = new HashSet<int>();
```

##### Visual Basic

```
Dim a As New HashSet(Of Integer)
```

The preceding line creates an empty integer set, [].

#### *HashSet(*T*) (IEnumerable(*T*))*

You can create a set with default values by passing an object that implements the *IEnumerable(*T*)* interface.

##### C#

```
HashSet<int> a = new HashSet<int>( new int [] { 2 , 4, 5 } );
```

##### Visual Basic

```
Dim a As New HashSet(Of Integer)(New Integer() {2, 4, 5})
```

This code creates the set [2,4,5].

### ***HashSet(T) ( IEqualityComparer(T))***

The constructor creates an empty *HashSet(T)* object with the specified comparer.

#### C#

```
HashSet<string> a = new HashSet<string>(StringComparer.CurrentCultureIgnoreCase);
```

#### Visual Basic

```
Dim a As New HashSet(Of String)(StringComparer.CurrentCultureIgnoreCase)
```

A *HashTable(string)* is created that ignores letter casing.

### ***HashSet(T) ( IEnumerable(T), IEqualityComparer(T))***

The constructor creates a *HashSet(T)* initialized with the specified items that uses the specified comparer.

#### C#

```
HashSet<string> a = new HashSet<string>(new string[] {"hi", "hola", "Hi"},  
    StringComparer.CurrentCultureIgnoreCase);
```

#### Visual Basic

```
Dim a As New HashSet(Of String)(New String() {"hi", "hola", "Hi"}, _  
    StringComparer.CurrentCultureIgnoreCase)
```

A *HashSet(string)* object is created containing the set *["hi", "hola"]*. The last *"hi"* is not added to the set because the *StringComparer.CurrentCultureIgnoreCase* was used for the comparer.

## **Adding Items to the *HashSet(T)***

To add items to the set, use the *Add* method.

### ***bool Add(T item)***

The *Add* method takes the item that you want to add as the argument and adds it to the set. The method returns *true* if the item is added to the set, or *false* if the item is already present in the set.

#### C#

```
HashSet<string> hs = new HashSet<string>(StringComparer.CurrentCultureIgnoreCase);  
  
hs.Add("hi");  
hs.Add("Hi");  
hs.Add("hola");
```

### Visual Basic

```
Dim hs As New HashSet(Of String)(StringComparer.CurrentCultureIgnoreCase)

hs.Add("hi")
hs.Add("Hi")
hs.Add("hola")
```

The set now contains *["hi", "hola"]*. *"Hi"* is not added to the set because *StringComparer.CurrentCultureIgnoreCase* states that *"hi"* and *"Hi"* are the same. Also notice how the *Add* method didn't throw an exception when the *"Hi"* was added.

## Removing Items from a *HashSet(T)*

You can remove items from a set by using the *Clear*, *Remove*, and *RemoveWhere* methods. None of these methods modify the capacity of the set. To modify the capacity of the set, you need to call the *TrimExcess* method, discussed later in this chapter.

### *void Clear()*

The *Clear* method removes all items from the list.

#### C#

```
HashSet<string> hs = new HashSet<string>(StringComparer.CurrentCultureIgnoreCase);

hs.Add("hi");
hs.Add("hola");

hs.Clear();
```

#### Visual Basic

```
Dim hs As New HashSet(Of String)(StringComparer.CurrentCultureIgnoreCase)

hs.Add("hi")
hs.Add("hola")

hs.Clear()
```

The set is empty, *[]*, after the *Clear* method is called. (But just as a reminder, the set still has the same capacity.)

### *bool Remove(T item)*

The *Remove* method removes a specified item from the set. The method returns *true* when the item is removed successfully, or *false* if the specified item either is not found or is not removed. Internally, the list locates items by using the equality comparer discussed in "Understanding the Equality and Ordering Comparers" section in Chapter 4, "Generic Collections."

**C#**

```
HashSet<int> hs = new HashSet<int>(new int[] { 1, 6, 2, 3 });
hs.Remove(6);
```

**Visual Basic**

```
Dim hs As New HashSet(Of Integer)(New Integer() {1, 6, 2, 3})
hs.Remove(6)
```

The preceding code changes the set from [ 1, 6, 2, 3 ] to [ 1, 2, 3 ] because a 6 was passed into the *Remove* method.

***int RemoveWhere(Predicate(T) match)***

*RemoveWhere* removes all items that match the specified predicate. The *Predicate(T)* is defined as follows.

**C#**

```
delegate bool Predicate<T>(T obj);
```

**Visual Basic**

```
Dim instance As New Predicate(Of T)(AddressOf HandlerMethod) as Boolean
```

*Predicate(T)* returns *true* when an object matches the criteria, meaning that it should be removed. The *RemoveWhere* method returns the number of items removed.

**Using a Method with RemoveWhere** The following code removes all even values by using a method for the *Predicate(T)*.

**C#**

```
static bool IsEven(int value)
{
    return (value % 2) == 0;
}
```

```
HashSet<int> hs = new HashSet<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
hs.RemoveWhere(IsEven);
```

**Visual Basic**

```
Function IsEven(ByVal value As Integer) As Boolean
    Return ((value Mod 2) = 0)
End Function
```

```
Dim hs As New HashSet(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})
hs.RemoveWhere(AddressOf IsEven)
```

**Using a Lambda Expression with RemoveWhere** The following code removes all even values by using a lambda expression for the *Predicate(T)*.

**C#**

```
HashSet<int> hs = new HashSet<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
hs.RemoveWhere(item => { return (item % 2) == 0; });
```

**Visual Basic**

```
Dim hs As New HashSet(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})
hs.RemoveWhere(Function(x) x Mod 2 = 0)
```

The set contains [ 1, 3, 5, 7, 9 ] after the *RemoveWhere* call completes.

## Performing Set Operations on a *HashSet(T)*

Set operations can be done using the *IntersectWith*, *IsProperSubsetOf*, *IsSubsetOf*, *IsProperSupersetOf*, *IsSupersetOf*, *Overlaps*, *UnionWith*, and *ExceptWith* methods.

### ***void IntersectWith(IEnumerable(T) other)***

This method keeps all items that are in the current set and the parameter set *other*. In other words, it removes all items that are not present in the set contained in the *other* parameter.

**C#**

```
HashSet<int> odds = new HashSet<int>(new int[] { 1, 3, 5, 7, 9 });
HashSet<int> primes = new HashSet<int>(new int[] { 2, 5, 7 });
HashSet<int> oddPrimes = new HashSet<int>(odds);

oddPrimes.IntersectWith(primes);
```

**Visual Basic**

```
Dim odds As New HashSet(Of Integer)(New Integer() {1, 3, 5, 7, 9})
Dim primes As New HashSet(Of Integer)(New Integer() {2, 5, 7})
Dim oddPrimes New HashSet(Of Integer)(odds)

oddPrimes.IntersectWith(primes)
```

The set *oddPrimes* contains {5,7}, because 5 and 7 are present in primes, {2,5,7}, and odds, {1,3,5,7,9}.

### ***bool IsProperSubsetOf(IEnumerable(T) other)***

This method checks whether the current set is a proper subset of the collection *other*. The method returns *true* if the current set is a subset of *other* and *other* contains elements that are not in the current set and both are not empty. The order doesn't matter.

**C#**

```
HashSet<int> numbers = new HashSet<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
HashSet<int> evens = new HashSet<int>(new int[] { 2, 4, 6, 8 });
HashSet<int> reversedNumbers = new HashSet<int>(new int[] { 9, 8, 7, 6, 5, 4, 3, 2, 1 });

Console.WriteLine("Evens {0} a proper subset of numbers.", 
    evens.IsProperSubsetOf(numbers) ? "is" : "is not");
Console.WriteLine("Reversednumbers {0} a proper subset of numbers.", 
    reversedNumbers.IsProperSubsetOf(numbers) ? "is" : "is not");
```

**Visual Basic**

```
Dim numbers As New HashSet(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})
Dim evens As New HashSet(Of Integer)(New Integer() {2, 4, 6, 8})
Dim reversedNumbers As New HashSet(Of Integer)(New Integer() {9, 8, 7, 6, 5, 4, 3, 2, 1})

Console.WriteLine("Evens {0} a proper subset of numbers.", _
    If(evens.IsProperSubsetOf(numbers), "is", "is not"))
Console.WriteLine("Reversednumbers {0} a proper subset of numbers.", _
    If(reversedNumbers.IsProperSubsetOf(numbers), "is", "is not"))
```

**Output**

Evens is a proper subset of numbers.  
Reversednumbers is not a proper subset of numbers.

Not all of the items in *numbers* are in *evens*, which makes *evens* a proper subset of numbers. The *reversedNumbers* variable contains the same items as *numbers*, which makes it not a proper subset.

***bool IsProperSupersetOf(IEnumerable(T) other)***

This method checks whether the current set is a proper superset of the collection *other*. The method returns *true* if the current set is a superset of *other* and the current set contains elements that are not in *other*; both are not empty. The order doesn't matter.

**C#**

```
HashSet<int> numbers = new HashSet<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
HashSet<int> evens = new HashSet<int>(new int[] { 2, 4, 6, 8 });
HashSet<int> reversedNumbers = new HashSet<int>(new int[] { 9, 8, 7, 6, 5, 4, 3, 2, 1 });

Console.WriteLine("Numbers {0} a proper superset of evens.", 
    numbers.IsProperSupersetOf(evens) ? "is" : "is not");
Console.WriteLine("Numbers {0} a proper superset of reversedNumbers.", 
    numbers.IsProperSupersetOf(reversedNumbers) ? "is" : "is not");
```

**Visual Basic**

```
Dim numbers As New HashSet(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})
Dim evens As New HashSet(Of Integer)(New Integer() {2, 4, 6, 8})
Dim reversedNumbers As New HashSet(Of Integer)(New Integer() {9, 8, 7, 6, 5, 4, 3, 2, 1})
```

```
Console.WriteLine("Numbers {0} a proper superset of evens.", _  
    If(numbers.IsProperSupersetOf(evens), "is", "is not"))  
Console.WriteLine("Numbers {0} a proper superset of reversedNumbers.", _  
    If(numbers.IsProperSupersetOf(reversedNumbers), "is", "is not"))
```

### Output

Numbers is a proper superset of evens.  
Numbers is not a proper superset of reversedNumbers.

Not all of the items in *numbers* are present in *evens*, which makes *numbers* a proper superset of *evens*. The *reversedNumbers* variable contains the same items as *numbers*, which makes it not a proper superset.

### ***bool IsSubsetOf(IEnumerable(T) other)***

This method checks whether the current set is a subset of the collection *other*. The method returns *true* if the current set is empty or if every element in the current set is present in *other*. This is the equivalent of saying *other.IsSupersetOf(this)*.

### C#

```
HashSet<int> evens = new HashSet<int>(new int[] { 2, 4, 6, 8 });  
HashSet<int> odds = new HashSet<int>(new int[] { 1, 3, 5, 7, 9 });  
HashSet<int> numbers = new HashSet<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });  
  
Console.WriteLine("Evens {0} a subset of numbers.",_  
    evens.IsSubsetOf(numbers) ? "are" : "are not");  
Console.WriteLine("Evens {0} a subset of odds.",_  
    evens.IsSubsetOf(odds) ? "are" : "are not");
```

### Visual Basic

```
Dim evens As New HashSet(Of Integer)(New Integer() {2, 4, 6, 8})  
Dim odds As New HashSet(Of Integer)(New Integer() {1, 3, 5, 7, 9})  
Dim numbers New HashSet(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})  
  
Console.WriteLine("Evens {0} a subset of numbers.", _  
    If(evens.IsSubsetOf(numbers), "are", "are not"))  
Console.WriteLine("Evens {0} a subset of odds.", _  
    If(evens.IsSubsetOf(odds), "are", "are not"))
```

### Output

Evens are a subset of numbers.  
Evens are not a subset of odds.

Every item in *evens* is present in *numbers*, which makes *evens* a subset of *numbers*. None of the items in *odd* is present in *evens* and vice versa, making neither a subset of the other.

### ***bool IsSupersetOf(IEnumerable(T) other)***

This method checks whether the current set is a superset of the collection *other*. The method returns *true* if the *other* is empty or if every element in *other* is present in the current set. This is the equivalent of saying *other*.*IsSubsetOf(this)*.

#### C#

```
HashSet<int> evens = new HashSet<int>(new int[] { 2, 4, 6, 8 });
HashSet<int> odds = new HashSet<int>(new int[] { 1, 3, 5, 7, 9 });
HashSet<int> numbers = new HashSet<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });

Console.WriteLine("Evens {0} a superset of numbers.", 
                  evens.IsSupersetOf(numbers) ? "are" : "are not");
Console.WriteLine("Numbers {0} a superset of odds.", 
                  numbers.IsSupersetOf(odds) ? "are" : "are not");
```

#### Visual Basic

```
Dim evens As New HashSet(Of Integer)(New Integer() {2, 4, 6, 8})
Dim odds As New HashSet(Of Integer)(New Integer() {1, 3, 5, 7, 9})
Dim numbers As New HashSet(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})

Console.WriteLine("Evens {0} a superset of numbers.", _
                  If(evens.IsSupersetOf(numbers), "are", "are not"))
Console.WriteLine("Numbers {0} a superset of odds.", _
                  If(numbers.IsSupersetOf(odds), "are", "are not"))
```

#### Output

```
Evens are not a superset of numbers.
Numbers are a superset of odds.
```

Every item in *numbers* is not present in *evens*, which makes *evens* not a superset of *numbers*. All of the items in *odd* are present in *numbers*, which makes *numbers* a superset of *odds*.

### ***bool Overlaps(IEnumerable(T) other)***

This method checks whether both collections, the current set and *other*, contain at least one of the same elements. It returns *true* when an item is present in both, *false* when it is not.

#### C#

```
HashSet<int> evens = new HashSet<int>(new int[] { 2, 4, 6, 8 });
HashSet<int> odds = new HashSet<int>(new int[] { 1, 3, 5, 7, 9 });
HashSet<int> numbers = new HashSet<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });

Console.WriteLine("Evens {0} with numbers.", 
                  evens.Overlaps(numbers) ? "overlaps" : "does not overlap");
Console.WriteLine("Evens {0} with odds.", 
                  evens.Overlaps(odds) ? "overlaps" : "does not overlap");
```

**Visual Basic**

```
Dim evens As New HashSet(Of Integer)(New Integer() {2, 4, 6, 8})
Dim odds As New HashSet(Of Integer)(New Integer() {1, 3, 5, 7, 9})
Dim numbers As New HashSet(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})

Console.WriteLine("Evens {0} with numbers.", _
                  If(evens.Overlaps(numbers), "overlaps", "does not overlap"))
Console.WriteLine("Evens {0} with odds.", _
                  If(evens.Overlaps(odds), "overlaps", "does not overlap"))
```

**Output**

Evens overlaps with numbers.  
Evens does not overlap with odds.

At least one item in *evens* is present in *numbers*. No items are present in both *evens* and *odds*.

***void UnionWith(IEnumerable(T) other)***

This method modifies the current set to contain all elements in itself and the parameter *other*.

**C#**

```
HashSet<int> evens = new HashSet<int>(new int[] { 2, 4, 6, 8 });
HashSet<int> odds = new HashSet<int>(new int[] { 1, 3, 5, 7, 9 });
HashSet<int> numbers = new HashSet<int>(odds);

numbers.UnionWith(evens);
```

**Visual Basic**

```
Dim evens As New HashSet(Of Integer)(New Integer() {2, 4, 6, 8})
Dim odds As New HashSet(Of Integer)(New Integer() {1, 3, 5, 7, 9})
Dim numbers As New HashSet(Of Integer)(odds)

numbers.UnionWith(evens)
```

The set *numbers* contains {1,3,5,7,9,2,4,6,8} after the set *odds*, {1,3,5,7,9}, is unioned with the set *evens*, {2,4,6,8}.

***void ExceptWith(IEnumerable(T) other)***

The *ExceptWith* method performs a set subtraction on the set. The operation removes all items that are in both collections, the current set and *other*.

**C#**

```
HashSet<int> odds = new HashSet<int>(new int[] { 1, 3, 5, 7, 9 });
HashSet<int> numbers = new HashSet<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
HashSet<int> evens = new HashSet<int>(numbers);

evens.ExceptWith(odds);
```

**C#**

```
Dim odds As New HashSet(Of Integer)(New Integer() {1, 3, 5, 7, 9})  
Dim numbers As New HashSet(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9})  
Dim evens As New HashSet(Of Integer)(numbers)  
  
evens.ExceptWith(odds)
```

The *ExceptWith* method removes all items that are in *odds*, making *evens* contain only even numbers, which in this case is {2,4,6,8}.

***bool Contains(T item)***

This method checks whether an item is present in the set. It returns *true* if the item is present in the set, or *false* if the item isn't present in the set.

**C#**

```
HashSet<int> evens = new HashSet<int>(new int[] { 2, 4, 6, 8 });  
  
Console.WriteLine("Evens {0} 1.", evens.Contains(1) ? "contains" : "doesn't contain");  
Console.WriteLine("Evens {0} 2.", evens.Contains(2) ? "contains" : "doesn't contain");
```

**Visual Basic**

```
Dim evens As New HashSet(Of Integer)(New Integer() {2, 4, 6, 8})  
  
Console.WriteLine("Evens {0} 1.", If(evens.Contains(1), "contains", "doesn't contain"))  
Console.WriteLine("Evens {0} 2.", If(evens.Contains(2), "contains", "doesn't contain"))
```

**Output**

```
Evens doesn't contain 1.  
Evens contains 2.
```

The number 2 is present in evens, but 1 is not.

***void TrimExcess()***

The *TrimExcess* method changes the capacity of the set. You need to call both *Clear* and then *TrimExcess* to restore a set to a condition similar to the original set created with the default constructor.

## Sorted Collections Overview

None of the collection classes that have been discussed so far sort automatically. You have to manually call the *Sort* routine on the *List(T)* class to sort it. Sorting some collection types, like the queue and stack, wouldn't make sense because it would be in violation of how they are designed to operate. To do automatic sorting, the .NET Framework has defined two classes named *SortedList(TKey, TValue)* and *SortedDictionary(TKey, TValue)*.

According to MSDN at [\*http://msdn.microsoft.com/en-us/library/f7fta44c\(v=VS.90\)\*](http://msdn.microsoft.com/en-us/library/f7fta44c(v=VS.90)):

*The SortedDictionary(TKey, TValue) generic class is a binary search tree with O(log n) retrieval, where n is the number of elements in the dictionary. In this respect, it is similar to the SortedList<(Of <(TKey, TValue)>) generic class. The two classes have similar object models, and both have O(log n) retrieval. Where the two classes differ is in memory use and speed of insertion and removal:*

*SortedList<(Of <(TKey, TValue)>) uses less memory than SortedDictionary(TKey, TValue).*

*SortedDictionary(TKey, TValue) has faster insertion and removal operations for unsorted data: O(log n) as opposed to O(n) for SortedList<(Of <(TKey, TValue)>)).*

*If the list is populated all at once from sorted data, SortedList<(Of <(TKey, TValue)>) is faster than SortedDictionary(TKey, TValue).*

### ***SortedList(TKey, TValue)***

The *SortedList(TKey, TValue)* class is basically the same as the *SortedDictionary(TKey, TValue)* class discussed in the next section. You should use the *SortedList(TKey, TValue)* class instead if memory is a concern or you are primary, dealing with presorted data. The *SortedList* class is the nongeneric version of this class.

### ***SortedDictionary(TKey, TValue)***

The difference between *SortedDictionary(TKey, TValue)* and *Dictionary(TKey, TValue)* is that the keys are sorted using the comparison operator in the *SortedDictionary(TKey, TValue)* but not in *Dictionary(TKey, TValue)*, as seen in the following code.

**C#**

```

Dictionary<string, int> unsorted = new Dictionary<string, int>();
unsorted.Add("one", 1);
unsorted.Add("two", 2);
unsorted.Add("three", 3);
unsorted.Add("four", 4);
unsorted.Add("five", 5);
unsorted.Add("six", 6);

Console.WriteLine("The unsorted list of keys are");
foreach (string name in unsorted.Keys)
{
    Console.WriteLine(name);
}

Console.WriteLine();

SortedDictionary<string, int> sorted = new SortedDictionary<string, int>();

sorted.Add("one", 1);
sorted.Add("two", 2);
sorted.Add("three", 3);
sorted.Add("four", 4);
sorted.Add("five", 5);
sorted.Add("six", 6);

Console.WriteLine("The sorted list of keys are");
foreach (string name in sorted.Keys)
{
    Console.WriteLine(name);
}

Console.WriteLine();

```

**Visual Basic**

```

Dim unsorted As New Dictionary(Of String, Integer)()
unsorted.Add("one", 1)
unsorted.Add("two", 2)
unsorted.Add("three", 3)
unsorted.Add("four", 4)
unsorted.Add("five", 5)
unsorted.Add("six", 6)

Console.WriteLine("The unsorted list of keys are")
For Each name As String In unsorted.Keys
    Console.WriteLine(name)
Next

Console.WriteLine()

Dim sorted As New SortedDictionary(Of String, Integer)()

sorted.Add("one", 1)
sorted.Add("two", 2)
sorted.Add("three", 3)

```

```
sorted.Add("four", 4)
sorted.Add("five", 5)
sorted.Add("six", 6)

Console.WriteLine("The sorted list of keys are")
For Each name As String In sorted.Keys
    Console.WriteLine(name)
Next

Console.WriteLine()
```

### Output

```
The unsorted list of keys are
one
two
three
four
five
six
```

```
The sorted list of keys are
five
four
one
six
three
two
```

As you can see, the keys are sorted in the *SortedDictionary(TKey, TValue)* but not in the *Dictionary(TKey, TValue)*. Other than that, you use the classes in the same way.



**More Info** *BindingList, IBindingList, IBindingListView, and BindingSource* are discussed in Chapter 10, "Using Collections with Windows Form Controls."

*ObservableCollection, INotifyCollectionChanged, ICollectionView, and CollectionView* are discussed in Chapter 11, "Using Collections with WPF and Silverlight Controls."

Synchronized collections are discussed briefly in Chapter 8. In Chapter 8, you learn how to use your custom classes in multiple threads as well as *SynchronizedCollection*, *SynchronizedKeyedCollection* and *SynchronizedReadOnlyCollection*.

## Summary

In this chapter, you saw how to use the built-in collections to create a queue, stack, hash set, and dictionary. You also saw how to create an array of bits or Booleans by using the *BitArray* class. The *DictionaryBase* and *CollectionBase* classes were briefly covered in case you run into some old code that uses them. You also saw how to use the sorted collections *SortedList(TKey, TValue)* and *SortedDictionary(TKey, TValue)*.



Part III

# Using Collections



# Chapter 6

# .NET Collection Interfaces

After completing this chapter, you will be able to

- Add enumeration interfaces to classes.
- Add collection interfaces to classes.
- Add list interfaces to classes.
- Add dictionary interfaces to classes.

## Enumerators (*IEnumerable* and *IEnumerator*) Overview

As you saw in Chapter 1, “Understanding Collections: Arrays and Linked Lists,” through Chapter 3, “Understanding Collections: Queues, Stacks, and Circular Buffers,” collections come in all shapes and sizes. They all have different ways that they can be manipulated and accessed. The way that each collection type interacts with the world is what makes each unique and useful. However, they all have one thing in common: they contain a collection of items. Traversing a stack, however, is done differently from traversing a list or dictionary. Your implementation of traversing a list might be different from your coworkers’ implementation. At some point, you may find it useful to generically traverse a collection regardless of its type and who created it. This is where the *IEnumerable*, *IEnumerator*, *IEnumerable(T)*, and *IEnumerator(T)* come into play.



**Note** The *IEnumerable(T)* and *IEnumerator(T)* interfaces are the generic interfaces to the *IEnumerable* and *IEnumerator*. All they do is add type-safe support. Both of them derive from their non-type-safe counterparts.

You can use the enumerator interfaces to iterate over a collection without needing to know the internal workings of the collection. The *IEnumerable* and *IEnumerable(T)* interfaces are used to define an object as being enumerable and are defined as follows.

C#

```
public interface IEnumerable
{
    I Enumerator GetEnumerator();
}

public interface IEnumerable<T> : IEnumerable
{
    I Enumerator<T> GetEnumerator();
}
```

**Visual Basic**

```

Public Interface IEnumerable(Of T)

    Inherits IEnumerable

    Function GetEnumerator() As IEnumerator(Of T)

End Interface

Public Interface IEnumerable

    Function GetEnumerator() As IEnumerator

End Interface

```

You can retrieve the enumerator of an object that exposes *IEnumerable* and/or *IEnumerable(T)* by calling the *GetEnumerator* method. All enumerator objects define the *IEnumerator* and/or *IEnumerator(T)* interface and are defined as follows.

**C#**

```

public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}

public interface IEnumerator<T> : IDisposable, IEnumerator
{
    T Current { get; }
}

```

**Visual Basic**

```

Public Interface IEnumerator
    Function MoveNext() As Boolean
    ReadOnly Property Current As Object
    Sub Reset()
End Interface

Public Interface IEnumerable(Of T)
    Inherits IDisposable, IEnumerator
    ReadOnly Property Current As T
End Interface

```

The *IEnumerator* and *IEnumerator(T)* interfaces let you traverse over a collection without knowing anything about it. The *Reset* method is only for COM interoperability.

Consider how the *foreach* statement uses the enumerator interfaces. The *Range* class, shown in the following example, defines a range of numbers to iterate over. The enumerator for the *Range* class writes to the console the name of the enumerator method or property that is

called. This lets you see how *foreach* interacts with the two interfaces. You can find the implementation of the class in the Chapter 6\CS\Criver\Range.cs file for C# or the Chapter 6\VB \Driver\Range.vb file for Microsoft Visual Basic.

### C#

```
Range range = new Range(10);

Console.WriteLine("starting");

foreach (int i in range)
{
    Console.WriteLine("\t\ti={0}", i);
}

Console.WriteLine("finished");
```

### Visual Basic

```
DevGuideToCollections.UnitTests.RunTests()

Dim range As New Range(10)

Console.WriteLine("starting")

For Each i As Integer In range
    Console.WriteLine(vbTab & vbTab & "i={0}", i)
Next

Console.WriteLine("finished")
```

### Output

```
starting
IEnumarable.GetEnumerator
    IEnumarator.MoveNext
    IEnumarator.Current
        i=1
    IEnumarator.MoveNext
    IEnumarator.Current
        i=2
    IEnumarator.MoveNext
    IEnumarator.Current
        i=3
    IEnumarator.MoveNext
    IEnumarator.Current
        i=4
    IEnumarator.MoveNext
    IEnumarator.Current
        i=5
    IEnumarator.MoveNext
    IEnumarator.Current
        i=6
    IEnumarator.MoveNext
    IEnumarator.Current
```

```

        i=7
IEnumator.MoveNext
IEnumator.Current
    i=8
IEnumator.MoveNext
IEnumator.Current
    i=9
IEnumator.MoveNext
IEnumator.Current
    i=10
IEnumator.MoveNext=false
finished

```

At the beginning of the loop, the *foreach* statement queries the *range* variable for the *IEnumerable(T)* interface. After it gets the interface, it calls the *GetEnumerator* method to obtain the enumerator for the *range* variable. The enumerator then calls the *MoveNext* method and gets the *Current* property until all items have been traversed. So from this, you can conclude that the *foreach* statement in the preceding code is the same as doing the following.

#### C#

```

if (range is IEnumerable<int>)
{
    IEnumator<int> enumerator = ((IEnumerable<int>)range).GetEnumerator();
    while (enumerator.MoveNext())
    {
        int i = enumerator.Current;
        Console.WriteLine("\t\ti={0}", i);
    }
}

```

#### Visual Basic

```

If (TypeOf range Is IEnumerable(Of Integer)) Then
    Dim enumerator As DirectCast(range, IEnumerable(Of Integer)).GetEnumerator()
    While (enumerator.MoveNext())
        Dim i As Integer = enumerator.Current
        Console.WriteLine(vbTab & vbTab & "i={0}", i)
    End While
End If

```



**Note** You should use the *foreach* statement instead of the preceding code. One advantage of using the *foreach* statement is that it lets your code use anything that may come out in the unforeseen future without you having to modify your code. Also, the *foreach* statement is a lot easier to read and understand.

## Adding Enumeration Support to Classes

Now you will see how to add enumeration support to your classes so that users can do a *foreach* and Language Integrated Query (LINQ) statement on your collections.



**More Info** LINQ is discussed in more detail in Chapter 7, “Introduction to LINQ.”

Enumeration support lets your class be used with some of the predefined functionality in the Microsoft .NET Framework, such as *AddRange* and constructors defined in some of the .NET collection classes as well as other classes. The following examples are implemented in partial classes to make it easier to follow along. The implemented interface name is contained in the file name of the file that contains the partial class. Add the keyword *partial* to each of the classes you created in Chapters 1 through 3.



**Note** You can follow along with the code in these sections by looking at the code in the Chapter 6\CS\DevGuideToCollections folder for C# and the Chapter 6\VB\DevGuideToCollections folder for Visual Basic.

So the following:

**C#**

```
public class ArrayEx<T>
```

**Visual Basic**

```
Public Class ArrayEx(Of T)
```

becomes:

**C#**

```
public partial class ArrayEx<T>
```

**Visual Basic**

```
Partial Public Class ArrayEx(Of T)
```

This lets you create a file that contains only the additional implementation. You can combine the code into one file, but it’s easier to follow along in the book by keeping each implementation in a separate file.

### ArrayEx(T)

First you need to add the *IEnumerable(T)* interface to the *ArrayEx(T)* class. Add an *ArrayEx.Enumerable.cs* class file or an *ArrayEx.Enumerable.vb* class file to the C# project or the Visual Basic project you created in Chapters 1 through 3.



**Note** Throughout this chapter, you're asked to add class files to the C# or Visual Basic project you worked with in Chapters 1 through 3. You create partial class files the same way you create class files. To create a class file, right-click the project in the Solution Explorer, click Add, and then click Class. From there, type the name of the class or the class file name. If you type the name of the class, Microsoft Visual Studio will create a file by using the name of the class and add a .cs or .vb extension to the file name, depending on whether you are in C# or Visual Basic. If you type a file name, Visual Studio will create a class named the same as the portion of your file name preceding the first period. For example, if you add a class and typed the file name ArrayEx.List.cs or ArrayEx.List.vb, Visual Studio would create a file with that name but the class within the file would be named ArrayEx (the portion of the name up to the first period).



**Note** You can find the completed source code in Samples\Chapter 6\##\DevGuideToCollections, where ## is either CS or VB.

Change the class declaration from the following:

**C#**

```
public class ArrayEx
```

**Visual Basic**

```
Public Class ArrayEx(Of T)
```

to this:

**C#**

```
public partial class ArrayEx<T> : IEnumerable<T>
```

**Visual Basic**

```
Partial Public Class ArrayEx(Of T)
    Implements IEnumerable(Of T)
```

The *IEnumerable(T)* interface contains a generic and nongeneric *GetEnumerator* method. You need to implement both methods. Both methods return the enumerator for this class, which you implement next as the *Enumerator* structure. The *Enumerator* structure takes a reference to the instance of the class being iterated in the constructor, so it has access to the data it needs to iterate through. The methods are defined as follows.

**C#**

```
/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
public IEnumerator<T> GetEnumerator()
{
    return new Enumerator(this);
}

/// <summary>
```

```
/// Gets the enumerator for this collection.  
/// </summary>  
/// <returns>The enumerator for this collection.</returns>  
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()  
{  
    return new Enumerator(this);  
}
```

### Visual Basic

```
'> <summary>  
'> Gets the enumerator for this collection.  
'> </summary>  
'> <returns>The enumerator for this collection.</returns>  
Public Function GetEnumerator() As IEnumerator(Of T) _  
    Implements IEnumerable(Of T).GetEnumerator  
    Return New Enumerator(Me)  
End Function  
  
'> <summary>  
'> Gets the enumerator for this collection.  
'> </summary>  
'> <returns>The enumerator for this collection.</returns>  
Private Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator  
    Return New Enumerator(Me)  
End Function
```

Next, the *Enumerator* structure needs to be implemented. The *Enumerator* structure derives from the *IEnumerator(T)* interface so that it can properly be used as an enumerator. The structure is defined as follows.

### C#

```
public struct Enumerator : IEnumerator<T>  
{  
    private ArrayEx<T> m_array;  
    private int m_index;  
    private int m_updateCode;  
  
    internal Enumerator(ArrayEx<T> array);  
    public T Current { get; }  
    public void Dispose();  
    object IEnumerator.Current { get; }  
    public bool MoveNext();  
    public void Reset();  
}
```

### Visual Basic

```
Public Structure Enumerator  
    Implements IEnumerator(Of T)  
  
    Private m_array As ArrayEx(Of T)  
    Private m_index As Integer  
    Private m_updateCode as Integer
```

```

Friend Sub New(ByVal array As ArrayEx(Of T))
Public ReadOnly Property Current As T
Public Sub Dispose() Implements IDisposable.Dispose
Private ReadOnly Property Current1 As Object
Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
Public Sub Reset() Implements IEnumerator.Reset
Private ReadOnly Property System.Collections.IEnumerator.Current As Object
End Structure

```

The *m\_array* field contains a reference to the *ArrayEx(T)* instance being iterated through. The *m\_index* field contains the index that you are currently on. The *m\_updateCode* is used to determine whether the collection has changed since the enumeration started.

The implementation of the constructor simply assigns *m\_array* field to the reference being passed in and sets the *m\_index* to -1, which is explained in the *MoveNext* method later in this section.

#### C#

```

internal Enumerator(ArrayEx<T> array)
{
    m_array = array;
    m_index = -1;
    m_updateCode = array.m_updateCode;
}

```

#### Visual Basic

```

Friend Sub New(ByVal array As ArrayEx(Of T))
    m_array = array
    m_index = -1
    m_updateCode = array.m_updateCode
End Sub

```

The *Dispose* method is defined in the *IDisposable* interface. Go to MSDN at <http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx> for help on how to use it.

#### C#

```

/// <summary>
/// Called when the resources should be released.
/// </summary>
public void Dispose()
{
}

```

#### Visual Basic

```

''' <summary>
''' Called when the resources should be released.
''' </summary>
Public Sub Dispose() Implements IDisposable.Dispose
End Sub

```

The *IEnumerator* structure implements the *IEnumerator* and *IEnumerator(T)* *Current* properties. Both implementations return the value at the current index by getting the element in *m\_array* at index *m\_index*.

### C#

```
/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
object System.Collections.IEnumerator.Current
{
    get { return m_array[m_index]; }

    /// <summary>
    /// Gets the element at the current position of the enumerator.
    /// </summary>
    public T Current
    {
        get { return m_array[m_index]; }
    }
}
```

### Visual Basic

```
'<summary>
'<summary> Gets the element at the current position of the enumerator.
'</summary>
Public ReadOnly Property Current() As T Implements IEnumrator(Of T).Current
    Get
        Return m_array(m_index)
    End Get
End Property

'<summary>
'<summary> Gets the element at the current position of the enumerator.
'</summary>
Private ReadOnly Property Current1() As Object Implements IEnumrator.Current
    Get
        Return m_array(m_index)
    End Get
End Property
```

The *MoveNext* method is implemented as follows.

### C#

```
/// <summary>
/// Advances the enumerator to the next element.
/// </summary>
/// <returns>
/// true if the enumerator was successfully advanced to the next element;
/// false if the enumerator has passed the end of the collection.
/// </returns>
```

```

public bool MoveNext()
{
    if (m_updateCode != m_array.m_updateCode)
    {
        throw new InvalidOperationException("The array was updated while traversing");
    }

    ++m_index;

    if (m_index >= m_array.Count)
    {
        return false;
    }

    return true;
}

```

### Visual Basic

```

''' <summary>
''' Advances the enumerator to the next element.
''' </summary>
''' <returns>
''' true if the enumerator was successfully advanced to the next element;
''' false if the enumerator has passed the end of the collection.
''' </returns>
Public Function MoveNext() As Boolean Implements IEnumerator(Of T).MoveNext
    If (m_updateCode <> m_array.m_updateCode) Then
        Throw New InvalidOperationException("The array was updated while traversing")
    End If

    m_index += 1

    If (m_index >= m_array.Count) Then
        Return False
    End If

    Return True
End Function

```

At the beginning of the “Adding Enumeration Support to Classes” section, you were shown how the .NET Framework calls the *IEnumerable(T).MoveNext* method immediately after the *IEnumerable(T).GetEnumerator* method. The *MoveNext* method always increments the index. Because *ArrayEx(T)* has a zero-based index, you need to make sure that *m\_index* is set to *0* after the first call. To do this, the *Reset* method and constructor sets *m\_index* to *-1* so that the first call of *MoveNext* sets *m\_index* to *0*. This eliminates having an extra variable to state that the enumerator has been reset or to start from the beginning.

The *Reset* method sets the *m\_index* field to *-1* as described in the preceding *MoveNext* code section and resets the *m\_updatecode*.

**C#**

```
/// <summary>
/// Sets the enumerator to its initial position,
/// which is before the first element in the collection.
/// </summary>
public void Reset()
{
    m_updateCode = m_array.m_updateCode;
    m_index = -1;
}
```

**Visual Basic**

```
'<summary>
'<summary> Sets the enumerator to its initial position,
'<summary> which is before the first element in the collection.
'<summary>
Public Sub Reset() Implements IEnumerator(Of T).Reset
    m_updateCode = m_array.m_updateCode
    m_index = -1
End Sub
```

## *CircularBuffer(T)*

First you need to add the *IEnumerable(T)* interface to the *CircularBuffer(T)* class. Add a CircularBuffer.Enumerable.cs class file or a CircularBuffer.Enumerable.vb class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

**C#**

```
public class CircularBuffer
```

**Visual Basic**

```
Public Class CircularBuffer(Of T)
```

to this:

**C#**

```
public partial class CircularBuffer<T> : IEnumerable<T>
```

**Visual Basic**

```
Partial Public Class CircularBuffer(Of T)
    Implements IEnumerable(Of T)
```

The *IEnumerable(T)* interface contains a generic and nongeneric *GetEnumerator* method. You need to implement both methods. Both methods return the enumerator for this class, which you implement next as the *Enumerator* structure. The *Enumerator* structure takes a reference to the instance of the class being iterated in the constructor, so it has access to the data it needs to iterate through. The methods are defined as follows.

**C#**

```

/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
public IEnumarator<T> GetEnumerator()
{
    return new Enumerator(this);
}

/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
System.Collections.IEnumarator System.Collections.IEnumerable.GetEnumerator()
{
    return new Enumerator(this);
}

```

**Visual Basic**

```

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Public Function GetEnumerator() As IEnumarator(Of T) _
    Implements IEnumarable(Of T).GetEnumerator
    Return New Enumerator(Me)
End Function

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Private Function GetEnumerator1() As IEnumarator Implements IEnumarable.GetEnumerator
    Return New Enumerator(Me)
End Function

```

Next, the *Enumerator* structure needs to be implemented. The *Enumerator* structure derives from the *IEnumerator(T)* interface so that it can be used properly as an enumerator. The structure is defined as follows.

**C#**

```

public struct Enumerator : IEnumarator<T>, IEnumarator
{
    private CircularBuffer<T> m_buffer;
    private int m_index;
    private int m_start;
    private int m_updateCode;
    internal Enumerator(CircularBuffer<T> buffer);
    public T Current { get; }
    public void Dispose();
    object IEnumarator.Current { get; }
}

```

```
    public bool MoveNext();
    public void Reset();
}
```

### Visual Basic

```
Public Structure Enumerator
    Implements IEnumerable(Of T)
    Private m_buffer As CircularBuffer(Of T)
    Private m_index As Integer
    Private m_start As Integer
    Private m_updateCode As Integer
    Friend Sub New(ByVal buffer As CircularBuffer(Of T))
        Public ReadOnly Property Current As T
        Public Sub Dispose() Implements IDisposable.Dispose
        Private ReadOnly Property Current1 As Object
        Public Function MoveNext() As Boolean Implements IEnumerable.MoveNext
        Public Sub Reset() Implements IEnumerable.Reset
        Private ReadOnly Property System.Collections.IEnumerator.Current As Object
    End Structure
```

The *m\_buffer* field contains a reference to the *CircularBuffer(T)* instance being iterated through. The *m\_index* field contains the index that you are currently on. The *m\_start* stores the index of the first valid element in the buffer. The *m\_updateCode* is used to determine whether the collection has changed since the enumeration started.

The implementation of the constructor simply assigns the *m\_buffer* field to the reference being passed in and sets the *m\_index* to -1, which is explained in the *MoveNext* method later in this section. The constructor also changes *m\_start* to the start of the buffer, which is explained in the *Current* property later in this section.

### C#

```
internal Enumerator(CircularBuffer<T> buffer)
{
    m_buffer = buffer;
    m_index = -1;
    m_start = m_buffer.m_start;
    m_updateCode = m_buffer.m_updateCode;
}
```

### Visual Basic

```
Friend Sub New(ByVal buffer As CircularBuffer(Of T))
    m_buffer = buffer
    m_index = -1
    m_start = m_buffer.m_start
    m_updateCode = m_buffer.m_updateCode
End Sub
```

The *Dispose* method is defined in the *IDisposable* interface. Go to MSDN at <http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx> for help on how to use it.

**C#**

```
/// <summary>
/// Called when the resources should be released.
/// </summary>
public void Dispose()
{
}
```

**Visual Basic**

```
''' <summary>
''' Called when the resources should be released.
''' </summary>
Public Sub Dispose() Implements IDisposable.Dispose
End Sub
```

The *IEnumerator* structure implements the *IEnumerator* and *IEnumerator(T)* *Current* properties. Both implementations return the value at the current index. Because a circular buffer is circular and its head can be anywhere in the array, you need a way of indexing the buffer from the head or *m\_start* as implemented in this structure. To do this, you can maintain an index called *m\_index* that you increment in the same way you did in the *ArrayEx(T)*.*Enumerator* structure. You can then add *m\_index* to *m\_start* to access the current element the enumerator has moved to. You can then use the modulus (mod) operation to handle indexing past the end of the array. So *m\_data[(m\_index + m\_start) % m\_buffer.m\_capacity]*, using C#, allows you to use *m\_index* as an index and start the index from the head of the buffer or *m\_start*. Doing a mod operation ensures that the value stays from 0 to (*m\_capacity* – 1) and properly moves *m\_index* to the beginning of the array when *m\_index* passes the end of the internal array.

**C#**

```
/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
public T Current
{
    get { return m_buffer.m_data[(m_index + m_start) % m_buffer.m_capacity]; }
}

/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
object System.Collections.IEnumerator.Current
{
    get { return m_buffer.m_data[(m_index + m_start) % m_buffer.m_capacity]; }
}
```

**Visual Basic**

```
''' <summary>
''' Gets the element at the current position of the enumerator.
''' </summary>
Public ReadOnly Property Current() As T Implements IEnumerator(Of T).Current
    Get
        Return m_buffer.m_data((m_index + m_start) Mod m_buffer.m_capacity)
    End Get

```

```
    End Get
End Property

''' <summary>
''' Gets the element at the current position of the enumerator.
''' </summary>
Private ReadOnly Property Current1() As Object Implements IEnumarator.Current
    Get
        Return m_buffer.m_data((m_index + m_start) Mod m_buffer.m_capacity)
    End Get
End Property
```

The *MoveNext* method works the same way as the *ArrayEx(T).Enumerator.MoveNext* method.

### C#

```
/// <summary>
/// Advances the enumerator to the next element.
/// </summary>
/// <returns>
/// true if the enumerator was successfully advanced to the next element;
/// false if the enumerator has passed the end of the collection.
/// </returns>
public bool MoveNext()
{
    if (m_updateCode != m_buffer.m_updateCode)
    {
        throw new InvalidOperationException("The array was updated while traversing");
    }

    ++m_index;

    if (m_index >= m_buffer.Count)
    {
        return false;
    }

    return true;
}
```

### Visual Basic

```
''' <summary>
''' Advances the enumerator to the next element.
''' </summary>
''' <returns>
''' true if the enumerator was successfully advanced to the next element;
''' false if the enumerator has passed the end of the collection.
''' </returns>
Public Function MoveNext() As Boolean Implements IEnumarator.MoveNext
    If (m_updateCode <> m_buffer.m_updateCode) Then
        Throw New InvalidOperationException("The array was updated while traversing")
    End If

    m_index += 1

    If (m_index >= m_buffer.Count) Then
```

```

        Return False
End If

        Return True
End Function

```

The *Reset* method sets the *m\_index* field to *-1* as described in the preceding *MoveNext* code section. The *m\_start* field is also set to the head of the buffer as explained in the *Current* property description earlier in this section.

#### C#

```

/// <summary>
/// Sets the enumerator to its initial position,
/// which is before the first element in the collection.
/// </summary>
public void Reset()
{
    m_index = -1;
    m_start = m_buffer.m_start;
    m_updateCode = m_buffer.m_updateCode;
}

```

#### Visual Basic

```

''' <summary>
''' Sets the enumerator to its initial position,
''' which is before the first element in the collection.
''' </summary>
Public Sub Reset() Implements IEnumerable.Reset
    m_index = -1
    m_start = m_buffer.m_start
    m_updateCode = m_buffer.m_updateCode
End Sub

```

## *SingleLinkedList(T)* and *DoubleLinkedList(T)*

Enumeration support in the *SingleLinkedList(T)* and *DoubleLinkedList(T)* classes is implemented the same way for both classes. First, you need to add the *IEnumerable(T)* interface to the *SingleLinkedList(T)* and *DoubleLinkedList(T)* classes. Add a *SingleLinkedList.Enumerable.cs* class file and a *DoubleLinkedList.Enumerable.cs* class file to the C# project, or a *SingleLinkedList.Enumerable.vb* file and a *DoubleLinkedList.Enumerable.vb* class file to the Visual Basic project you worked with in Chapters 1 through 3.

Change the *SingleLinkedList(T)* class declaration from the following:

#### C#

```
public class SingleLinkedList
```

#### Visual Basic

```
Public Class SingleLinkedList(Of T)
```

to this:

**C#**

```
public partial class SingleLinkedList<T> : IEnumerable<T>
```

**Visual Basic**

```
Partial Public Class SingleLinkedList(Of T)
    Implements IEnumerable(Of T)
```

And change the *DoubleLinkedList(T)* class from the following:

**C#**

```
public class DoubleLinkedList
```

**Visual Basic**

```
Public Class DoubleLinkedList(Of T)
```

to this:

**C#**

```
public partial class DoubleLinkedList<T> : IEnumerable<T>
```

**Visual Basic**

```
Partial Public Class DoubleLinkedList(Of T)
    Implements IEnumerable(Of T)
```

The *IEnumerable(T)* interface contains a generic and nongeneric *GetEnumerator* method. You need to implement both methods. Both methods return the enumerator for this class, which you implement next as the *Enumerator* structure. The *Enumerator* structure takes a reference to the instance of the class being iterated in the constructor, so it has access to the data it needs to iterate through. The methods are defined as follows.

**C#**

```
/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
public IEnumerator<T> GetEnumerator()
{
    return new Enumerator(this);
}

/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return new Enumerator(this);
}
```

### Visual Basic

```
' <summary>
' Gets the enumerator for this collection.
' </summary>
' <returns>The enumerator for this collection.</returns>
Public Function GetEnumerator() As IEnumerator(Of T) _
    Implements IEnumerable(Of T).GetEnumerator
    Return New Enumerator(Me)
End Function

' <summary>
' Gets the enumerator for this collection.
' </summary>
' <returns>The enumerator for this collection.</returns>
Private Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator
    Return New Enumerator(Me)
End Function
```

Next, the *Enumerator* structure needs to be implemented. The *Enumerator* structure derives from the *IEnumerator(T)* interface so that it can be used properly as an enumerator.

For the *SingleLinkedList(T)* class, the structure is defined as follows.

### C#

```
public struct Enumerator : IEnumerator<T>
{
    private SingleLinkedList<T> m_list;
    private SingleLinkedListNode<T> m_current;
    private bool m_end;
    internal Enumerator(SingleLinkedList<T> list);
    public T Current { get; }
    public void Dispose();
    object IEnumerator.Current { get; }
    public bool MoveNext();
    public void Reset();
}
```

### Visual Basic

```
Public Structure Enumerator
    Implements IEnumerator(Of T)
    Private m_list As SingleLinkedList(Of T)
    Private m_current As SingleLinkedListNode(Of T)
    Private m_end As Boolean
    Private m_updateCode as Integer
    Friend Sub New(ByVal list As SingleLinkedList(Of T))
        Public ReadOnly Property Current As T
        Public Sub Dispose() Implements IDisposable.Dispose
        Private ReadOnly Property Current1 As Object
        Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
        Public Sub Reset() Implements IEnumerator.Reset
        Private ReadOnly Property System.Collections.IEnumerator.Current As Object
    End Structure
```

For the *DoubleLinkedList(T)* class, the structure is defined as follows.

**C#**

```
public struct Enumerator : IEnumerator<T>
{
    private DoubleLinkedList<T> m_list;
    private DoubleLinkedListNode<T> m_current;
    private bool m_end;
    private int m_updateCode;
    internal Enumerator(DoubleLinkedList<T> list);
    public T Current { get; }
    public void Dispose();
    object IEnumerator.Current { get; }
    public bool MoveNext();
    public void Reset();
}
```

### Visual Basic

```
Public Structure Enumerator
    Implements IEnumerable(Of T)
    Private m_list As DoubleLinkedList(Of T)
    Private m_current As DoubleLinkedListNode(Of T)
    Private m_end As Boolean
    Private m_updateCode As Integer
    Friend Sub New(ByVal list As DoubleLinkedList(Of T))
        Public ReadOnly Property Current As T
        Public Sub Dispose() Implements IDisposable.Dispose
        Private ReadOnly Property Current1 As Object
        Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
        Public Sub Reset() Implements IEnumerator.Reset
        Private ReadOnly Property System.Collections.IEnumerator.Current As Object
    End Structure
```

The *m\_list* field contains a reference to the *SingleLinkedList(T)* or *DoubleLinkedList(T)* instance being iterated through. The *m\_current* field contains the current node that the enumerator is on. The *m\_end* field states whether the enumerator has reached the end of the list. The *m\_updateCode* is used to determine whether the collection has changed since the enumeration started.

The implementation of the constructor simply assigns *m\_list* field to the reference being passed in and sets the *m\_current* to *null* and *m\_end* to *false*, which is explained in the *MoveNext* method later in this section.

For the *SingleLinkedList(T)* enumerator, use the following.

**C#**

```
internal Enumerator(SingleLinkedList<T> list)
{
    m_list = list;
    m_current = null;
    m_end = false;
    m_updateCode = list.m_updateCode;
}
```

**Visual Basic**

```
Friend Sub New(ByVal list As SingleLinkedList(Of T))
    m_list = list
    m_current = Nothing
    m_end = False
    m_updateCode = list.m_updateCode
End Sub
```

For the *DoubleLinkedList(T)* enumerator, use the following.

**C#**

```
internal Enumerator(DoubleLinkedList<T> list)
{
    m_list = list;
    m_current = null;
    m_end = false;
    m_updateCode = list.m_updateCode;
}
```

**Visual Basic**

```
Friend Sub New(ByVal list As DoubleLinkedList(Of T))
    m_list = list
    m_current = Nothing
    m_end = False
    m_updateCode = list.m_updateCode
End Sub
```

The *Dispose* method is defined in the *IDisposable* interface. Go to MSDN at <http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx> for help on how to use it.

**C#**

```
/// <summary>
/// Called when the resources should be released.
/// </summary>
public void Dispose()
{}
```

**Visual Basic**

```
' <summary>
' Called when the resources should be released.
' </summary>
Public Sub Dispose() Implements IDisposable.Dispose
End Sub
```

The *Enumerator* structure implements the *IEnumerator* and *IEnumerator(T)* *Current* properties. Both implementations return the value at the current node by getting the value in *m\_current*.

**C#**

```
/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
```

```
public T Current
{
    get { return m_current.Data; }
}

/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
object System.Collections.IEnumerator.Current
{
    get { return m_current.Data; }
}
```

### Visual Basic

```
'> <summary>
'> Gets the element at the current position of the enumerator.
'> </summary>
Public ReadOnly Property Current() As T Implements IEnumerable(Of T).Current
    Get
        Return m_current.Data
    End Get
End Property

'> <summary>
'> Gets the element at the current position of the enumerator.
'> </summary>
Private ReadOnly Property Current1() As Object Implements IEnumerator.Current
    Get
        Return m_current.Data
    End Get
End Property
```

The *MoveNext* method is implemented as follows.

### C#

```
/// <summary>
/// Advances the enumerator to the next element.
/// </summary>
/// <returns>
/// true if the enumerator was successfully advanced to the next element;
/// false if the enumerator has passed the end of the collection.
/// </returns>
public bool MoveNext()
{
    if (m_updateCode != m_list.m_updateCode)
    {
        throw new InvalidOperationException("The list was updated while traversing");
    }

    if (m_end || m_list.IsEmpty)
    {
        return false;
    }

    if (m_current == null)
```

```

{
    m_current = m_list.Head;
}
else if (m_current == m_list.Tail)
{
    m_end = true;
    m_current = null;
    return false;
}

else
{
    m_current = m_current.Next;
}

return true;
}

```

**Visual Basic**

```

''' <summary>
''' Advances the enumerator to the next element.
''' </summary>
''' <returns>
''' true if the enumerator was successfully advanced to the next element;
''' false if the enumerator has passed the end of the collection.
''' </returns>
Public Function MoveNext() As Boolean Implements IEnumrator(Of T).MoveNext
    If (m_updateCode <> m_list.m_updateCode) Then
        Throw New InvalidOperationException("The list was updated while traversing")
    End If

    If (m_end Or m_list.IsEmpty) Then
        Return False
    End If

    If (m_current Is Nothing) Then
        m_current = m_list.Head
    ElseIf (m_current Is m_list.Tail) Then
        m_end = True
        m_current = Nothing
        Return False
    Else
        m_current = m_current.Next
    End If

    Return True
End Function

```

At the beginning of the "Adding Enumeration Support to Classes" section earlier in this chapter, you saw how the .NET Framework calls the *IEnumerable(T).MoveNext* method immediately after the *IEnumerable(T).GetEnumerator* method. You need to ensure that the very first *MoveNext* call moves *m\_current* to the beginning of the linked list. You will know that you should start at the beginning of the list when *m\_current* is equal to *null*. If on a call to

*MoveNext* you discover you are on the last node, you can set *m\_end* to *true* and return *false* to state that you have moved past the end of the list.

The *Reset* method sets the *m\_current* to *null* and *m\_end* to *false* as described in the *MoveNext* code earlier in this section.

#### C#

```
/// <summary>
/// Sets the enumerator to its initial position,
/// which is before the first element in the collection.
/// </summary>
public void Reset()
{
    m_current = null;
    m_end = false;
    m_updateCode = m_list.m_updateCode;
}
```

#### Visual Basic

```
''' <summary>
''' Sets the enumerator to its initial position,
''' which is before the first element in the collection.
''' </summary>
Public Sub Reset() Implements IEnumerator(Of T).Reset
    m_current = Nothing
    m_end = False
    m_updateCode = m_list.m_updateCode
End Sub
```

## QueuedArray(*T*)

First, you need to add the *IEnumerable(T)* interface to the *QueuedArray(T)* class. Add a *QueuedArray.Enumerable.cs* class file or a *QueuedArray.Enumerable.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

#### C#

```
public class QueuedArray
```

#### Visual Basic

```
Public Class QueuedArray(Of T)
```

to this:

#### C#

```
public partial class QueuedArray<T> : IEnumerable<T>
```

### Visual Basic

```
Partial Public Class QueuedArray(Of T)
    Implements IEnumerable(Of T)
```

The *IEnumerable(T)* interface contains a generic and nongeneric *GetEnumerator* method. You need to implement both methods. Both methods return the enumerator for this class, which you implement next as the *Enumerator* structure. The *Enumerator* structure takes a reference to the instance of the class being iterated in the constructor, so it has access to the data it needs to iterate through. The methods are defined as follows.

### C#

```
/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
public IEnumerator<T> GetEnumerator()
{
    return new Enumerator(this);
}

/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return new Enumerator(this);
}
```

### Visual Basic

```
''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Public Function GetEnumerator() As IEnumerable(Of T) _
    Implements IEnumerable(Of T).GetEnumerator
    Return New Enumerator(Me)
End Function

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Private Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator
    Return New Enumerator(Me)
End Function
```

Next, the *Enumerator* structure needs to be implemented. The *Enumerator* structure derives from the *IEnumerator(T)* interface so that it can properly be used as an enumerator. The structure is defined as follows.

**C#**

```
public struct Enumerator : IEnumerator<T>, IEnumerator
{
    private QueuedArray<T> m_buffer;
    private int m_index;
    private int m_head;
    private int m_updateCode;
    internal Enumerator(QueuedArray<T> buffer);
    public T Current { get; }
    public void Dispose();
    object IEnumerator.Current { get; }
    public bool MoveNext();
    public void Reset();
}
```

**Visual Basic**

```
Public Structure Enumerator
    Implements IEnumerator(Of T)
    Private m_buffer As QueuedArray(Of T)
    Private m_index As Integer
    Private m_head As Integer
    Private m_updateCode As Integer
    Friend Sub New(ByVal buffer As QueuedArray(Of T))
        Public ReadOnly Property Current As T
        Public Sub Dispose() Implements IDisposable.Dispose
        Private ReadOnly Property Current1 As Object
        Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
        Public Sub Reset() Implements IEnumerator.Reset
        Private ReadOnly Property System.Collections.IEnumerator.Current As Object
    End Structure
```

The *m\_buffer* field contains a reference to the *QueuedArray(T)* instance being iterated through. The *m\_index* field contains the index that you are currently on. The *m\_head* field stores the index of the first valid element in the queue. The *m\_updateCode* is used to determine whether the collection has changed since the enumeration started.

The implementation of the constructor simply assigns the *m\_buffer* field to the reference being passed in and sets the *m\_index* to -1, which is explained in the *MoveNext* property later in this section. The constructor also sets *m\_head* to the head of the buffer, which is explained in the *Current* property later in this section.

**C#**

```
internal Enumerator(QueuedArray<T> buffer)
{
    m_buffer = buffer;
    m_index = -1;
    m_head = m_buffer.m_head;
    m_updateCode = m_buffer.m_updateCode;
}
```

**Visual Basic**

```
Friend Sub New(ByVal buffer As QueuedArray(Of T))
    m_buffer = buffer
    m_index = -1
    m_head = m_buffer.m_head
    m_updateCode = m_buffer.m_updateCode
End Sub
```

The *Dispose* method is defined in the *IDisposable* interface. Go to MSDN at <http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx> for help on how to use it.

**C#**

```
/// <summary>
/// Called when the resources should be released.
/// </summary>
public void Dispose()
{}
```

**Visual Basic**

```
''' <summary>
''' Called when the resources should be released.
''' </summary>
Public Sub Dispose() Implements IDisposable.Dispose
End Sub
```

The *Enumerator* structure implements the *IEnumerator* and *IEnumerator(T)* *Current* properties. Both implementations return the value at the current index. *QueuedArray(T)* uses the internal array like the circular buffer uses its internal array, so the head can be anywhere in the array. You need a way of indexing the buffer from the head that is stored in this structure. To do this, you can maintain an index called *m\_index* that you increment in the same way you did in the *ArrayEx(T).Enumerator* structure. You can then add *m\_index* to *m\_head* to access the current element the enumerator has moved to. You can then use the mod operation to handle indexing past the end of the array. So *m\_data[(m\_index + m\_head) % m\_buffer.m\_data.Length]* (using C#) allows you to use *m\_index* as an index and starts the index from the head of the array. Doing a mod ensures that the value stays from 0 to (*m\_data.Length* – 1) and properly moves *m\_index* to the beginning of the array when it reaches the end.

**C#**

```
/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
public T Current
{
    get { return m_buffer.m_data[(m_index + m_head) % m_buffer.m_data.Length]; }
}

/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
```

```
object System.Collections.IEnumerator.Current
{
    get { return m_buffer.m_data[(m_index + m_head) % m_buffer.m_data.Length]; }
}
```

### Visual Basic

```
''<summary>
''' Gets the element at the current position of the enumerator.
''' </summary>
Public ReadOnly Property Current() As T Implements IEnumerator(Of T).Current
    Get
        Return m_buffer.m_data((m_index + m_head) Mod m_buffer.m_data.Length)
    End Get
End Property

''' <summary>
''' Gets the element at the current position of the enumerator.
''' </summary>
Private ReadOnly Property Current1() As Object Implements IEnumerator(Of T).Current
    Get
        Return m_buffer.m_data((m_index + m_head) Mod m_buffer.m_data.Length)
    End Get
End Property
```

The *MoveNext* method works the same way as the *ArrayEx(T).Enumerator.MoveNext* method.

### C#

```
///<summary>
/// Advances the enumerator to the next element.
/// </summary>
/// <returns>
/// true if the enumerator was successfully advanced to the next element;
/// false if the enumerator has passed the end of the collection.
/// </returns>
public bool MoveNext()
{
    if (m_updateCode != m_buffer.m_updateCode)
    {
        throw new InvalidOperationException("The queue was updated while traversing");
    }

    ++m_index;

    if (m_index >= m_buffer.Count)
    {
        return false;
    }

    return true;
}
```

### Visual Basic

```

''' <summary>
''' Advances the enumerator to the next element.
''' </summary>
''' <returns>
''' true if the enumerator was successfully advanced to the next element;
''' false if the enumerator has passed the end of the collection.
''' </returns>
Public Function MoveNext() As Boolean Implements IEnumator.MoveNext
    If (m_updateCode <> m_buffer.m_updateCode) Then
        Throw New InvalidOperationException("The queue was updated while traversing")
    End If

    m_index += 1

    If (m_index >= m_buffer.Count) Then
        Return False
    End If

    Return True
End Function

```

The *Reset* method sets the *m\_index* field to *-1*, as described in the preceding *MoveNext* method. The *m\_head* field is also set to the head of the buffer, as described in the *Current* property earlier in this section.

### C#

```

/// <summary>
/// Sets the enumerator to its initial position,
/// which is before the first element in the collection.
/// </summary>
public void Reset()
{
    m_index = -1;
    m_head = m_buffer.m_head;
    m_updateCode = m_buffer.m_updateCode;
}

```

### Visual Basic

```

''' <summary>
''' Sets the enumerator to its initial position,
''' which is before the first element in the collection.
''' </summary>
Public Sub Reset() Implements IEnumator.Reset
    m_index = -1
    m_head = m_buffer.m_head
    m_updateCode = m_buffer.m_updateCode
End Sub

```

## QueuedLinkedList(*T*)

First you need to add the *IEnumerable(T)* interface to the *QueuedLinkedList(T)* class. Add a *QueuedLinkedList.Enumerable.cs* class file or a *QueuedLinkedList.Enumerable.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

**C#**

```
public class QueuedLinkedList
```

**Visual Basic**

```
Public Class QueuedLinkedList(Of T)
```

to this:

**C#**

```
public partial class QueuedLinkedList<T> : IEnumerable<T>
```

**Visual Basic**

```
Partial Public Class QueuedLinkedList(Of T)
    Implements IEnumerable(Of T)
```

The *IEnumerable(T)* interface contains a generic and nongeneric *GetEnumerator* method. Items are iterated in a queue in the same order as items are iterated in a linked list. So, you can return the enumerator of the internal data storage, which is implemented as a linked list, instead of creating an enumerator for the queue.

**C#**

```
/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
public IEnumerator<T> GetEnumerator()
{
    return m_data.GetEnumerator();
}
/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return m_data.GetEnumerator();
}
```

**Visual Basic**

```

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Public Function GetEnumerator() As IEnumerator(Of T) _
    Implements IEnumerable(Of T).GetEnumerator
    Return m_data.GetEnumerator()
End Function

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Private Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator
    Return m_data.GetEnumerator()
End Function

```

Because the *GetEnumerator* method returns the enumerator of the internal data storage, you don't need to implement an *Enumerator* structure.

***StackedArray(T)***

First you need to add the *IEnumerable(T)* interface to the *StackedArray(T)* class. Add a *StackedArray.Enumerable.cs* class file or a *StackedArray.Enumerable.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

**C#**

```
public class StackedArray
```

**Visual Basic**

```
Public Class StackedArray(Of T)
```

to this:

**C#**

```
public partial class StackedArray<T> : IEnumerable<T>
```

**Visual Basic**

```
Partial Public Class StackedArray(Of T)
    Implements IEnumerable(Of T)
```

The *IEnumerable(T)* interface contains a generic and nongeneric *GetEnumerator* method. You need to implement both methods. Both methods return the enumerator for this class, which you implement as the *Enumerator* structure next. The *Enumerator* structure takes a reference to the instance of the class being iterated in the constructor, so it has access to the data it needs to iterate through. The methods are defined as follows.

### C#

```
/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
public IEnumarator<T> GetEnumerator()
{
    return new Enumerator(this);
}

/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
System.Collections.IEnumarator System.Collections.IEnumerable.GetEnumerator()
{
    return new Enumerator(this);
}
```

### Visual Basic

```
''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Public Function GetEnumerator() As IEnumarator(Of T) _
    Implements IEnumarable(Of T).GetEnumerator
    Return New Enumerator(Me)
End Function

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Private Function GetEnumerator1() As IEnumarator Implements IEnumarable.GetEnumerator
    Return New Enumerator(Me)
End Function
```

Next the *Enumerator* structure needs to be implemented. The *Enumerator* structure derives from the *IEnumarator(T)* interface so that it can be used properly as an enumerator. The structure is defined as follows.

**C#**

```
public struct Enumerator : IEnumerator<T>
{
    private StackedArray<T> m_array;
    private int m_index;
    private int m_updateCode;

    internal Enumerator(StackedArray<T> array);
    public T Current { get; }
    public void Dispose();
    object IEnumerator.Current { get; }
    public bool MoveNext();
    public void Reset();
}
```

**Visual Basic**

```
Public Structure Enumerator
    Implements IEnumerator(Of T)
    Private m_array As StackedArray(Of T)
    Private m_index As Integer
    Private m_updateCode As Integer
    Friend Sub New(ByVal array As StackedArray(Of T))
        Public ReadOnly Property Current As T
        Public Sub Dispose() Implements IDisposable.Dispose
        Public ReadOnly Property Current1 As Object
        Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
        Public Sub Reset() Implements IEnumerator.Reset
        Public ReadOnly Property System.Collections.IEnumerator.Current As Object
    End Structure
```

The *m\_array* field contains a reference to the *StackedArray(T)* instance being iterated through. The *m\_index* field contains the index that you are currently on. The *m\_updateCode* is used to determine whether the collection has changed since the enumeration started.

The implementation of the constructor simply assigns the *m\_array* field to the reference being passed in and sets the *m\_index* to *m\_array.Count*, which is explained in the *MoveNext* method later in this section.

**C#**

```
internal Enumerator(ArrayEx<T> array)
{
    m_array = array;
    m_index = m_array.Count;
    m_updateCode = m_array.m_data.UpdateCode;
}
```

**Visual Basic**

```
Friend Sub New(ByVal array As StackedArray(Of T))
    m_array = array
    m_index = m_array.Count
    m_updateCode = m_array.m_data.UpdateCode
End Sub
```

The *Dispose* method is defined in the *IDisposable* interface. Go to MSDN at <http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx> for help on how to use it.

### C#

```
/// <summary>
/// Called when the resources should be released.
/// </summary>
public void Dispose()
{
}
```

### Visual Basic

```
'<summary>
'Called when the resources should be released.
'</summary>
Public Sub Dispose() Implements IDisposable.Dispose
End Sub
```

The *Enumerator* structure implements the *IEnumerator* and *IEnumerator(T)* *Current* properties. Both implementations return the value at the current index by getting the element in *m\_array* at index *m\_index*.

### C#

```
/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
object System.Collections.IEnumerator.Current
{
    get { return m_array[m_index]; }
}

/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
public T Current
{
    get { return m_array[m_index]; }
}
```

### Visual Basic

```
'<summary>
'Gets the element at the current position of the enumerator.
'</summary>
Public ReadOnly Property Current() As T Implements Ienumerator(Of T).Current
    Get
        Return m_array.m_data(m_index)
    End Get
End Property

'<summary>
'Gets the element at the current position of the enumerator.
'</summary>
```

```

Public ReadOnly Property Current1() As Object Implements IEnumerable.Current
    Get
        Return m_array.m_data(m_index)
    End Get
End Property

```

The *MoveNext* method is implemented as follows.

#### C#

```

/// <summary>
/// Advances the enumerator to the next element.
/// </summary>
/// <returns>
/// true if the enumerator was successfully advanced to the next element;
/// false if the enumerator has passed the end of the collection.
/// </returns>
public bool MoveNext()
{
    if (m_updateCode != m_array.m_data.UpdateCode)
    {
        throw new InvalidOperationException("The stack was updated while traversing");
    }

    --m_index;

    if (m_index <= 0)
    {
        return false;
    }

    return true;
}

```

#### Visual Basic

```

''' <summary>
''' Advances the enumerator to the next element.
''' </summary>
''' <returns>
''' true if the enumerator was successfully advanced to the next element;
''' false if the enumerator has passed the end of the collection.
''' </returns>
Public Function MoveNext() As Boolean Implements IEnumerable.MoveNext
    If (m_updateCode <> m_array.m_data.UpdateCode) Then
        Throw New InvalidOperationException("The stack was updated while traversing")
    End If

    m_index -= 1

    If (m_index <= 0) Then
        Return False
    End If

    Return True
End Function

```

At the beginning of the "Adding Enumeration Support to Classes" section, you saw how the .NET Framework calls the *IEnumerator(T).MoveNext* method immediately after the *IEnumerable(T).GetEnumerator* method. The *MoveNext* method always decrements the index. Stacks are accessed in Last In, First Out (LIFO) order. So the iterator needs to go in reverse order if you want to iterate over the collection in the order items are removed. Because *ArrayEx(T)* has a zero-based index, you need to ensure that *m\_index* is set to (*m\_array.Count* – 1) after the first call. To do this, the *Reset* method and constructor set *m\_index* to *m\_array.Count* so that the first call to *MoveNext* sets it to (*m\_array.Count* – 1). This eliminates having an extra variable to state that the enumerator has been reset or to start from the beginning.

The *Reset* method sets the *m\_index* field to *m\_array.Count* as described in the *MoveNext* code earlier in this section.

#### C#

```
/// <summary>
/// Sets the enumerator to its initial position,
/// which is before the first element in the collection.
/// </summary>
public void Reset()
{
    m_index = m_array.Count;
    m_updateCode = m_array.m_data.UpdateCode;
}
```

#### Visual Basic

```
''' <summary>
''' Sets the enumerator to its initial position,
''' which is before the first element in the collection.
''' </summary>
Public Sub Reset() Implements IEnumerator.Reset
    m_index = m_array.Count
    m_updateCode = m_array.m_data.UpdateCode
End Sub
```

## *StackedLinkedList(T)*

First you need to add the *IEnumerable(T)* interface to the *StackedLinkedList(T)* class. Add a *StackedLinkedList.Enumerable.cs* class file or a *StackedLinkedList.Enumerable.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

#### C#

```
public class StackedLinkedList
```

#### Visual Basic

```
Public Class StackedLinkedList(Of T)
```

to this:

**C#**

```
public partial class StackedLinkedList<T> : IEnumerable<T>
```

**Visual Basic**

```
Partial Public Class StackedLinkedList(Of T)
    Implements IEnumerable(Of T)
```

The *IEnumerable(T)* interface contains a generic and nongeneric *GetEnumerator* method. You need to implement both methods. Both methods return the enumerator for this class, which you implement next as the *Enumerator* structure. The *Enumerator* structure takes a reference to the instance of the class being iterated in the constructor, so it has access to the data it needs to iterate through. The methods are defined as follows.

**C#**

```
/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
public IEnumerator<T> GetEnumerator()
{
    return new Enumerator(this);
}

/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return new Enumerator(this);
}
```

**Visual Basic**

```
''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Public Function GetEnumerator() As IEnumerator(Of T) _
    Implements IEnumerable(Of T).GetEnumerator
    Return New Enumerator(Me)
End Function

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Private Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator
    Return New Enumerator(Me)
End Function
```

Next the *Enumerator* structure needs to be implemented. The *Enumerator* structure derives from the *IEnumerator(T)* interface so that it can be used properly as an enumerator. The structure is defined as follows.

### C#

```
public struct Enumerator : Ienumerator<T>
{
    private StackedLinkedList<T> m_list;
    private DoubleLinkedListNode<T> m_current;
    private bool m_end;
    private int m_updateCode;
    internal Enumerator(StackedLinkedList<T> list);
    public T Current { get; }
    public void Dispose();
    object Ienumerator.Current { get; }
    public bool MoveNext();
    public void Reset();
}
```

### Visual Basic

```
Public Structure Enumerator
    Implements Ienumerator(Of T)
    Private m_list As StackedLinkedList(Of T)
    Private m_current As DoubleLinkedListNode(Of T)
    Private m_end As Boolean
    Private m_updateCode As Integer
    Friend Sub New(ByVal list As StackedLinkedList(Of T))
        Public ReadOnly Property Current As T
        Public Sub Dispose() Implements IDisposable.Dispose
        Public ReadOnly Property Current1 As Object
        Public Function MoveNext() As Boolean Implements Ienumerator.MoveNext
        Public Sub Reset() Implements Ienumerator.Reset
        Public ReadOnly Property System.Collections.Ienumerator.Current As Object
    End Structure
```

The *m\_list* field contains a reference to the *StackedLinkedList(T)* instance being iterated through. The *m\_current* field contains the current node the enumerator is on. The *m\_end* field states whether the enumerator has reached the end of the list. The *m\_updateCode* is used to determine whether the collection has changed since the enumeration started.

The implementation of the constructor simply assigns *m\_list* field to the reference being passed in and sets the *m\_current* to *null* and *m\_end* to *false*, which is explained in the *MoveNext* method later in this section.

### C#

```
internal Enumerator(StackedLinkedList<T> list)
{
    m_list = list;
    m_current = null;
    m_end = false;
    m_updateCode = list.m_data.UpdateCode;
}
```

**Visual Basic**

```
Friend Sub New(ByVal list As StackedLinkedList(Of T))
    m_list = list
    m_current = Nothing
    m_end = False
    m_updateCode = list.m_data.UpdateCode
End Sub
```

The *Dispose* method is defined in the *IDisposable* interface. Go to MSDN at <http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx> for help on how to use it.

**C#**

```
/// <summary>
/// Called when the resources should be released.
/// </summary>
public void Dispose()
{}
```

**Visual Basic**

```
''' <summary>
''' Called when the resources should be released.
''' </summary>
Public Sub Dispose() Implements IDisposable.Dispose
End Sub
```

The *Enumerator* structure implements the *IEnumerator* and *IEnumerator(T)* *Current* properties. Both implementations return the value at the current node by getting the value in *m\_current*.

**C#**

```
/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
public T Current
{
    get { return m_current.Data; }
}

/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
object System.Collections.IEnumerator.Current
{
    get { return m_current.Data; }
}
```

**Visual Basic**

```
''' <summary>
''' Gets the element at the current position of the enumerator.
''' </summary>
```

```
Public ReadOnly Property Current() As T Implements IEnumerable(Of T).Current
    Get
        Return m_current.Data
    End Get
End Property

''' <summary>
''' Gets the element at the current position of the enumerator.
''' </summary>
Public ReadOnly Property Current1() As Object Implements IEnumerator.Current
    Get
        Return m_current.Data
    End Get
End Property
```

The *MoveNext* method is implemented as follows.

### C#

```
/// <summary>
/// Advances the enumerator to the next element.
/// </summary>
/// <returns>
/// true if the enumerator was successfully advanced to the next element;
/// false if the enumerator has passed the end of the collection.
/// </returns>
public bool MoveNext()
{
    if (m_updateCode != m_list.m_data.UpdateCode)
    {
        throw new InvalidOperationException("The stack was updated while traversing");
    }

    if (m_end || m_list.IsEmpty)
    {
        return false;
    }

    if (m_current == null)
    {
        m_current = m_list.m_data.Tail;
    }
    else if (m_current == m_list.m_data.Head)
    {
        m_end = true;
        m_current = null;
        return false;
    }
    else
    {
        m_current = m_current.Previous;
    }
}

return true;
}
```

### Visual Basic

```

''' <summary>
''' Advances the enumerator to the next element.
''' </summary>
''' <returns>
''' true if the enumerator was successfully advanced to the next element;
''' false if the enumerator has passed the end of the collection.
''' </returns>
Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
    If (m_updateCode <> m_list.m_data.UpdateCode) Then
        Throw New InvalidOperationException("The stack was updated while traversing")
    End If

    If (m_end Or m_list.IsEmpty) Then
        Return False
    End If

    If (m_current Is Nothing) Then
        m_current = m_list.m_data.Tail
    ElseIf (m_current Is m_list.m_data.Head) Then
        m_end = True
        m_current = Nothing
        Return False
    Else
        m_current = m_current.Previous
    End If

    Return True
End Function

```

At the beginning of the “Adding Enumeration Support to Classes” section, you saw how the .NET Framework calls the *IEnumerator(T).MoveNext* method immediately after the *IEnumerable(T).GetEnumerator* method. Also, stacks are LIFO collection types, so you need to iterate from the end of the list to the beginning. To do that, you need to ensure that the very first *MoveNext* call moves *m\_current* to the end of the linked list. You will know that you should start at the end of the list when *m\_current* is equal to *null*. If on a call to *MoveNext* you discover that you are on the first node, you can set *m\_end* to *true* and return *false* to state that you have moved past the beginning of the list.

The *Reset* method sets the *m\_current* to *null* and *m\_end* to *false* as described in the *MoveNext* method earlier in this section.

### C#

```

/// <summary>
/// Sets the enumerator to its initial position,
/// which is before the first element in the collection.
/// </summary>
public void Reset()
{
    m_updateCode = m_list.m_data.UpdateCode;
    m_current = null;
    m_end = false;
}

```

**Visual Basic**

```
''' <summary>
''' Sets the enumerator to its initial position,
''' which is before the first element in the collection.
''' </summary>
Public Sub Reset() Implements IEnumerator.Reset
    m_updateCode = m_list.m_data.UpdateCode
    m_current = Nothing
    m_end = False
End Sub
```

## **AssociativeArrayAL(TKey,TValue)**

First you need to add the *IEnumerable(T)* interface to the *AssociativeArrayAL(TKey,TValue)* class. Add an *AssociativeArrayAL.Enumerable.cs* class file or an *AssociativeArrayAL.Enumerable.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

**C#**

```
public class AssociativeArrayAL
```

**Visual Basic**

```
Public Class AssociativeArrayAL(Of TKey, TValue)
```

to this:

**C#**

```
public partial class AssociativeArrayAL<TKey, TValue> : IEnumerable<
KeyValuePair<TKey, TValue> >
```

**Visual Basic**

```
Partial Public Class AssociativeArrayAL(Of TKey, TValue)
    Implements IEnumerable(Of KeyValuePair(Of TKey, TValue))
```

The *IEnumerable(T)* interface contains a generic and nongeneric *GetEnumerator* method. You need to implement both methods. Both methods return the enumerator for this class, which you implement next as the *Enumerator* structure. The *Enumerator* structure takes a reference to the instance of the class being iterated in the constructor, so it has access to the data it needs to iterate through. The methods are defined as follows.

**C#**

```
/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
public IEnumerator<T> GetEnumerator()
{
    return new Enumerator(this);
}
```

```

/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return new Enumerator(this);
}

```

### Visual Basic

```

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Public Function GetEnumerator() As IEnumerator(Of KeyValuePair(Of TKey, TValue)) _
    Implements IEnumerable(Of KeyValuePair(Of TKey, TValue)).GetEnumerator
    Return New Enumerator(Me)
End Function

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Private Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator
    Return New Enumerator(Me)
End Function

```

Next the *Enumerator* structure needs to be implemented. The *Enumerator* structure derives from the *IEnumerator(T)* interface so that it can be used properly as an enumerator. The structure will be defined as follows.

### C#

```

public struct Enumerator : IEnumerator<KeyValuePair<TKey, TValue>>
{
    private AssociativeArrayList<TKey, TValue> m_aa;
    private DoubleLinkedListNode<AssociativeArrayList<TKey, TValue>.KVPair> m_currentNode;
    private KeyValuePair<TKey, TValue> m_current;
    private int m_updateCode;
    private bool m_end;
    internal Enumerator(AssociativeArrayList<TKey, TValue> aa);
    public KeyValuePair<TKey, TValue> Current { get; }
    public void Dispose();
    object IEnumerator.Current { get; }
    public bool MoveNext();
    public void Reset();
}

```

### Visual Basic

```
Public Structure Enumerator
    Implements IEnumerable(Of KeyValuePair(Of TKey, TValue))
    Private m_aa As AssociativeArrayAL(Of TKey, TValue)
    Private m_currentNode As DoubleLinkedListNode(Of KVPair(Of TKey, TValue))
    Private m_current As KeyValuePair(Of TKey, TValue)
    Private m_updateCode As Integer
    Private m_end As Boolean
    Friend Sub New(ByVal aa As AssociativeArrayAL(Of TKey, TValue))
        Public ReadOnly Property Current As KeyValuePair(Of TKey, TValue)
        Public Sub Dispose() Implements IDisposable.Dispose
        Public ReadOnly Property Current1 As Object
        Public Function MoveNext() As Boolean Implements IEnumerable.MoveNext
        Public Sub Reset() Implements IEnumerable.Reset
        Public ReadOnly Property System.Collections.IEnumerator.Current As Object
    End Structure
```

The *m\_aa* field contains a reference to the *AssociativeArrayAL(TKey,TValue)* instance being iterated through. The *m\_currentNode* field contains the current node the enumerator is on. The *m\_end* field states whether the enumerator has reached the end of the list. The *m\_updateCode* stores the current update code. The *m\_current* field stores the current key/value pair.

The implementation of the constructor simply assigns the *m\_aa* field to the reference being passed in and sets the *m\_currentNode* to null and *m\_end* to *false*, which is explained in the *MoveNext* code later in this section.

### C#

```
internal Enumerator(AssociativeArrayAL<TKey, TValue> aa)
{
    m_aa = aa;

    m_current = default(KeyValuePair<TKey, TValue>);
    m_updateCode = m_aa.m_updateCode;
    m_currentNode = null;
    m_end = false;
}
```

### Visual Basic

```
Friend Sub New(ByVal aa As AssociativeArrayAL(Of TKey, TValue))
    m_aa = aa

    m_current = CType(Nothing, KeyValuePair(Of TKey, TValue))
    m_updateCode = m_aa.m_updateCode
    m_currentNode = Nothing
    m_end = False
End Sub
```

The *Dispose* method is defined in the *IDisposable* interface. Go to MSDN at <http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx> for help on how to use it.

### C#

```
/// <summary>
/// Called when the resources should be released.
/// </summary>
public void Dispose()
{
}
```

### Visual Basic

```
''' <summary>
''' Called when the resources should be released.
''' </summary>
Public Sub Dispose() _
    Implements IEnumerator<_
        Of System.Collections.Generic.KeyValuePair(Of TKey, TValue) _> _
    ).Dispose
End Sub
```

The *IEnumerator* structure implements the *IEnumerator* and *IEnumerator(T)* *Current* properties. Both implementations return the value at the current index by returning *m\_current*.

### C#

```
/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
public T Current
{
    get { return m_current; }
}

/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
object System.Collections.IEnumerator.Current
{
    get { return m_current; }
}
```

### Visual Basic

```
''' <summary>
''' Gets the element at the current position of the enumerator.
''' </summary>
Public ReadOnly Property Current() As KeyValuePair(Of TKey, TValue) Implements _
    IEnumerator<_
        Of System.Collections.Generic.KeyValuePair(Of TKey, TValue) _> _
    ).Current
    Get
        Return m_current
    End Get
End Property
```

```
''' <summary>
''' Gets the element at the current position of the enumerator.
''' </summary>
Public ReadOnly Property Current1() As Object Implements IEnumrator.Current
    Get
        Return m_current
    End Get
End Property
```

The *MoveNext* method is implemented as follows.

C#

```
/// <summary>
/// Advances the enumerator to the next element.
/// </summary>
/// <returns>
/// true if the enumerator was successfully advanced to the next element;
/// false if the enumerator has passed the end of the collection.
/// </returns>
public bool MoveNext()
{
    if (m_updateCode != m_aa.m_updateCode)
    {
        throw new InvalidOperationException("The hash table was updated while traversing");
    }

    if (m_end)
    {
        return false;
    }

    if (m_currentNode == null)
    {
        if (m_aa.IsEmpty)
        {
            m_end = true;
            return false;
        }
    }

    m_currentNode = m_aa.m_list.Head;

    KVPair kvp = m_currentNode.Data;
    m_current = new KeyValuePair<TKey, TValue>(kvp.Key, kvp.Value);

    return true;
}

m_currentNode = m_currentNode.Next;

if (m_currentNode == null)
{
    m_end = true;
    return false;
}
```

```

    m_current = new KeyValuePair< TKey, TValue>
        (m_currentNode.Data.Key, m_currentNode.Data.Value);

    return true;
}

```

### Visual Basic

```

''' <summary>
''' Advances the enumerator to the next element.
''' </summary>
''' <returns>
''' true if the enumerator was successfully advanced to the next element;
''' false if the enumerator has passed the end of the collection.
''' </returns>
Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
    If (m_updateCode <> m_aa.m_updateCode) Then
        Throw New InvalidOperationException("The hash table was updated while traversing")
    End If

    If (m_end) Then
        Return False
    End If

    If (m_currentNode Is Nothing) Then
        If (m_aa.IsEmpty) Then
            m_end = True
            Return False
        End If

        m_currentNode = m_aa.m_list.Head

        Dim kvp As KVPair = m_currentNode.Data
        m_current = New KeyValuePair(Of TKey, TValue)(kvp.Key, kvp.Value)

        Return True
    End If

    m_currentNode = m_currentNode.Next

    If (m_currentNode Is Nothing) Then
        m_end = True
        Return False
    End If

    m_current = New KeyValuePair(Of TKey, TValue) _
        (m_currentNode.Data.Key, m_currentNode.Data.Value)

    Return True
End Function

```

The *MoveNext* method operates similarly to the *MoveNext* method of the *CircularBuffer(T).Enumerator* structure. (See the section titled “*SingleLinkedList(T)* and *DoubleLinkedList(T)*” earlier in this chapter for more information.) The update code is used to see whether the

collection has changed since the traversing started. This eliminates an unknown state the collection could be in if it is rehashed or items are removed or inserted.

The *Reset* method sets the *m\_currentNode* to *null* and *m\_end* to *false* as described in the *MoveNext* method earlier in this section.

#### C#

```
/// <summary>
/// Sets the enumerator to its initial position,
/// which is before the first element in the collection.
/// </summary>
public void Reset()
{
    m_currentNode = null;
    m_end = false;
    m_updateCode = m_aa.m_updateCode;
    m_current = default(KeyValuePair< TKey, TValue>);
}
```

#### Visual Basic

```
''' <summary>
''' Sets the enumerator to its initial position,
''' which is before the first element in the collection.
''' </summary>
Public Sub Reset() Implements IEnumerator.Reset
    m_currentNode = Nothing
    m_end = False
    m_updateCode = m_aa.m_updateCode
    m_current = CType(Nothing, KeyValuePair(Of TKey, TValue))
End Sub
```

## **AssociativeArrayHT(TKey,TValue)**

First you need to add the *IEnumerable(T)* interface to the *AssociativeArrayHT(TKey,TValue)* class. Add an *AssociativeArrayHT.Enumerable.cs* class file or an *AssociativeArrayHT.Enumerable.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

#### C#

```
public class AssociativeArrayHT
```

#### Visual Basic

```
Public Class AssociativeArrayHT(Of TKey, TValue)
```

to this:

#### C#

```
public partial class AssociativeArray< TKey, TValue> : IEnumerable< KeyValuePair< TKey, TValue>>
```

### Visual Basic

```
Partial Public Class AssociativeArrayHT(Of TKey, TValue)
    Implements IEnumerable(Of KeyValuePair(Of TKey, TValue))
```

The *IEnumerable(T)* interface contains a generic and nongeneric *GetEnumerator* method. You need to implement both methods. Both methods return the enumerator for this class, which you implement next as the *Enumerator* structure. The *Enumerator* structure takes a reference to the instance of the class being iterated in the constructor, so it has access to the data it needs to iterate through. The methods are defined as follows.

### C#

```
/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
public IEnumerator<T> GetEnumerator()
{
    return new Enumerator(this);
}

/// <summary>
/// Gets the enumerator for this collection.
/// </summary>
/// <returns>The enumerator for this collection.</returns>
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return new Enumerator(this);
}
```

### Visual Basic

```
''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Public Function GetEnumerator() As IEnumerator(Of KeyValuePair(Of TKey, TValue)) _
    Implements IEnumerable(Of KeyValuePair(Of TKey, TValue)).GetEnumerator
    Return New Enumerator(Me)
End Function

''' <summary>
''' Gets the enumerator for this collection.
''' </summary>
''' <returns>The enumerator for this collection.</returns>
Private Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator
    Return New Enumerator(Me)
End Function
```

Next the *Enumerator* structure needs to be implemented. The *Enumerator* structure derives from the *IEnumerator(T)* interface so that it can be used properly as an enumerator. The structure is defined as follows.

**C#**

```
public struct Enumerator : IEnumerator<KeyValuePair<TKey, TValue>>
{
    private AssociativeArrayHT<TKey, TValue> m_aa;
    private int m_entryIndex;
    private int m_bucketIndex;
    private int m_updateCode;
    private KeyValuePair<TKey, TValue> m_current;
    internal Enumerator(AssociativeArrayHT<TKey, TValue> aa);
    public KeyValuePair<TKey, TValue> Current { get; }
    public void Dispose();
    object IEnumerator.Current { get; }
    public bool MoveNext();
    public void Reset();
}
```

**Visual Basic**

```
Public Structure Enumerator
    Implements IEnumerable(Of KeyValuePair(Of TKey, TValue))
    Private m_aa As AssociativeArrayHT(Of TKey, TValue)
    Private m_entryIndex As Integer
    Private m_bucketIndex As Integer
    Private m_updateCode As Integer
    Private m_current As KeyValuePair(Of TKey, TValue)
    Friend Sub New(ByVal aa As AssociativeArrayHT(Of TKey, TValue))
        Public ReadOnly Property Current As KeyValuePair(Of TKey, TValue)
        Public Sub Dispose() Implements IDisposable.Dispose
        Public ReadOnly Property Current1 As Object
        Public Function MoveNext() As Boolean Implements IEnumerable.MoveNext
        Public Sub Reset() Implements IEnumerable.Reset
        Public ReadOnly Property System.Collections.IEnumerator.Current As Object
    End Structure
```

The *m\_aa* field contains a reference to the *AssociativeArrayHT(T)* instance being iterated through. The *m\_current* field contains the current key/value pair the enumerator is on. The *m\_entryIndex* and *m\_bucketIndex* contain the entry and bucket the index is on. The *m\_updateCode* stores a snapshot of the update code when the traversing beings. The implementation of the constructor simply assigns the *m\_aa* field to the reference being passed in and sets the *m\_bucketIndex* and *m\_entryIndex* to *NULL\_REFERENCE*, which is explained in the *MoveNext* property later in this section.

**C#**

```
internal Enumerator(AssociativeArrayHT<TKey, TValue> aa)
{
    m_current = default(KeyValuePair<TKey, TValue>);
    m_aa = aa;
    m_updateCode = m_aa.m_updateCode;
    m_bucketIndex = NULL_REFERENCE;
    m_entryIndex = NULL_REFERENCE;
}
```

**Visual Basic**

```

Friend Sub New(ByVal aa As AssociativeArrayHT(Of TKey, TValue))
    m_current = CType(Nothing, KeyValuePair(Of TKey, TValue))
    m_aa = aa
    m_updateCode = m_aa.m_updateCode
    m_bucketIndex = NULL_REFERENCE
    m_entryIndex = NULL_REFERENCE
End Sub

```

The *Dispose* method is defined in the *IDisposable* interface. Go to MSDN at <http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx> for help on how to use it.

**C#**

```

/// <summary>
/// Called when the resources should be released.
/// </summary>
public void Dispose()
{
}

```

**Visual Basic**

```

''' <summary>
''' Called when the resources should be released.
''' </summary>
Public Sub Dispose() Implements IEnumerator(Of KeyValuePair(Of TKey, TValue)).Dispose
End Sub

```

The *IEnumerator* structure implements the *IEnumerator* and *IEnumerator(T)* *Current* properties. Both implementations return the value at the current index by returning *m\_current*.

**C#**

```

/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
public T Current
{
    get { return m_current; }
}

/// <summary>
/// Gets the element at the current position of the enumerator.
/// </summary>
object System.Collections.IEnumerator.Current
{
    get { return m_current; }
}

```

**Visual Basic**

```

''' <summary>
''' Gets the element at the current position of the enumerator.
''' </summary>

```

```

Public ReadOnly Property Current() As KeyValuePair(Of TKey, TValue) Implements _
    IEnumarator<_
        Of System.Collections.Generic.KeyValuePair(Of TKey, TValue) _ 
    >.Current
    Get
        Return m_current
    End Get
End Property

''' <summary>
''' Gets the element at the current position of the enumerator.
''' </summary>
Public ReadOnly Property Current1() As Object Implements IEnumarator.Current
    Get
        Return m_current
    End Get
End Property

```

The *MoveNext* method is implemented as follows.

### C#

```

/// <summary>
/// Advances the enumerator to the next element.
/// </summary>
/// <returns>
/// true if the enumerator was successfully advanced to the next element;
/// false if the enumerator has passed the end of the collection.
/// </returns>
public bool MoveNext()
{
    if (m_updateCode != m_aa.m_updateCode)
    {
        throw new InvalidOperationException("The hash table was updated while traversing");
    }

    if (m_aa.MoveNext(ref m_bucketIndex, ref m_entryIndex))
    {
        m_current = new KeyValuePair<TKey, TValue>
            (m_aa.m_entries[m_entryIndex].Key, m_aa.m_entries[m_entryIndex].Value);
        return true;
    }

    return false;
}

```

### Visual Basic

```

''' <summary>
''' Advances the enumerator to the next element.
''' </summary>
''' <returns>
''' true if the enumerator was successfully advanced to the next element;
''' false if the enumerator has passed the end of the collection.
''' </returns>

```

```

Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
    If (m_updateCode <> m_aa.m_updateCode) Then
        Throw New InvalidOperationException("The hash table was updated while traversing")
    End If

    If (m_aa.MoveNext(m_bucketIndex, m_entryIndex)) Then
        m_current = New KeyValuePair(Of TKey, TValue)_
            (m_aa.m_entries(m_entryIndex).Key, m_aa.m_entries(m_entryIndex).Value)
        Return True
    End If

    Return False
End Function

```

At the beginning of the "Adding Enumeration Support to Classes" section, you saw how the .NET Framework calls the *IEnumerator(T).MoveNext* method immediately after the *IEnumerable(T).GetEnumerator* method. The *MoveNext* method calls the *AssociativeArrayHT(TKey,TValue).MoveNext* method. This method is described in the "Adding Helper Methods and Properties" sections in Chapter 2, "Understanding Collections: Associative Arrays."

The *Reset* method sets the *m\_bucketIndex* and *m\_entryIndex* to *NULL\_REFERENCE* as described in the *AssociativeArrayHT(TKey,TValue).MoveNext* method in Chapter 2.

### C#

```

/// <summary>
/// Sets the enumerator to its initial position, which is before the
/// first element in the collection.
/// </summary>
public void Reset()
{
    m_bucketIndex = NULL_REFERENCE;
    m_updateCode = m_aa.m_updateCode;
    m_entryIndex = NULL_REFERENCE;
    m_current = new KeyValuePair<TKey, TValue>();
}

```

### Visual Basic

```

''' <summary>
''' Sets the enumerator to its initial position, which is before the
''' first element in the collection.
''' </summary>
Public Sub Reset() Implements IEnumerator.Reset
    m_bucketIndex = NULL_REFERENCE
    m_updateCode = m_aa.m_updateCode
    m_entryIndex = NULL_REFERENCE
    m_current = CType(Nothing, KeyValuePair(Of TKey, TValue))
End Sub

```

## ***ICollection and ICollection(T) Overview***

The .NET Framework defines two interfaces called *ICollection* and *ICollection(T)* for defining an object as a collection. The interfaces are not interchangeable and are not derived from each other.

The *ICollection* interface adds size, enumerator, and synchronization support to a collection type. All of the collection classes are derived from this interface. The following is the interface definition.

### C#

```
public interface ICollection : System.Collections.IEnumerable
{
    void CopyTo(System.Array array, int index);
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
}
```

### Visual Basic

```
Public Interface ICollection
    Inherits IEnumerable
    Sub CopyTo(ByVal array As Array, ByVal index As Integer)
    ReadOnly Property Count As Integer
    ReadOnly Property IsSynchronized As Boolean
    ReadOnly Property SyncRoot As Object
End Interface
```

The *ICollection(T)* interface defines generic collection manipulation support for a collection. The interface has support for adding, removing, and checking the contents of a collection, as you can see from the following interface definition.

### C#

```
public interface ICollection<T> : System.Collections.Generic.IEnumerable<T>
{
    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    int Count { get; }
    bool IsReadOnly { get; }
    bool Remove(T item);
}
```

### Visual Basic

```
Public Interface ICollection(Of T)
    Inherits IEnumerable(Of T)
    Sub Add(ByVal item As T)
    Sub Clear()
    Function Contains(ByVal item As T) As Boolean
    Sub CopyTo(ByVal array As T(), ByVal arrayIndex As Integer)
    Function Remove(ByVal item As T) As Boolean
    ReadOnly Property Count As Integer
    ReadOnly Property IsReadOnly As Boolean
End Interface
```

Some collection types, such as stack and queue, have problems implementing the *ICollection(T)* interface because the user cannot specifically state where they want to add an item or what item they want to remove. However, all collections do have a size and can be copied to an array. Read-only collections are a special type of collection that cannot have items removed from them or added to them. The *IsReadOnly* property is used to state whether a collection is read-only. The *CopyTo* method lets you copy the contents of the collection to an array starting at the specified index of the array.

So, you can use the *ICollection* interface to do basic collection operations such as getting the size of the collection, copying it to an array, and threading support with the *IsSynchronized* and *SyncRoot* properties that are discussed in Chapter 8, "Using Threads with Collections." With *ICollection(T)*, you can do generic collection manipulations such as removals and additions as well as get the size of the collection, check the contents of it, and copy the contents.

## Adding Collection Support to Classes

Giving your collection types .NET collection support is useful so that other users can interact with them in the same manner as other .NET collection. To do this, you need to add *ICollection* support to the classes you created in Chapters 1 through 3, and add *ICollection(T)* to some of them.



**Note** You can follow along with the code in these sections by using the code in the DevGuideToCollections project in the Chapter 6\CS\DevGuideToCollections folder for C# or the Chapter 6\VB\DevGuideToCollections folder for Visual Basic.

### ArrayEx(*T*)

First you need to add the *ICollection(T)* and *ICollection* interfaces to the *ArrayEx(T)* class. Add an *ArrayEx.Collection.cs* class file or an *ArrayEx.Collection.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

**C#**

```
public class ArrayEx
```

**Visual Basic**

```
Public Class ArrayEx(Of T)
```

to this:

**C#**

```
public partial class ArrayEx<T> : ICollection<T>, System.Collection.ICollection
```

**Visual Basic**

```
Partial Public Class ArrayEx(Of T)
    Implements ICollection(Of T), ICollection
```

All of the methods and properties except *CopyTo*, *IsReadOnly*, *IsSynchronized*, and *SyncRoot* are already implemented.

The *IsSynchronized* and *SyncRoot* properties are discussed in detail in Chapter 8, so for now you'll implement them as follows.

**C#**

```
bool System.Collections.ICollection.IsSynchronized
{
    get { return false; }
}

object System.Collections.ICollection.SyncRoot
{
    get { throw new NotImplementedException(); }
}
```

**Visual Basic**

```
Private ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized
    Get
        Return False
    End Get
End Property

Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property
```

Because the *ArrayEx(T)* can be modified, the *IsReadOnly* property needs to return *false*.

**C#**

```
/// <summary>
/// Gets a value indicating whether the collection is read only.
/// </summary>
public bool IsReadOnly
{
    get { return false; }
}
```

**Visual Basic**

```
'<summary>
'Gets a value indicating whether the collection is read only.
'</summary>
Public ReadOnly Property IsReadOnly() As Boolean Implements ICollection(Of T).IsReadOnly
    Get
        Return False
    End Get
End Property
```

There isn't anything special about the array in *ArrayEx(T)*, so the *CopyTo* methods can use the *Array.Copy* to method. You need to copy only the elements that are actually used, so the *Count* property instead of the *Capacity* property has to be passed to the *Array.Copy* method as well.

**C#**

```
public void CopyTo(T[] array, int arrayIndex)
{
    Array.Copy(m_data, 0, array, arrayIndex, m_count);
}

void System.Collections.ICollection.CopyTo(Array array, int index)
{
    Array.Copy(m_data, 0, array, index, m_count);
}
```

**Visual Basic**

```
'<summary>
'Copies the elements of the array to the specified array.
'</summary>
'<param name="array">The array to copy the data to.</param>
'<param name="arrayIndex">The index in array where copying begins.</param>
Public Sub CopyTo(ByVal array As T(), ByVal arrayIndex As Integer) _
    Implements ICollection(Of T).CopyTo
    System.Array.Copy(m_data, 0, array, arrayIndex, m_count)
End Sub

Private Sub CopyTo1(ByVal array As Array, ByVal index As Integer) _
    Implements ICollection.CopyTo
    System.Array.Copy(m_data, 0, array, index, m_count)
End Sub
```

The *Implements* statement is shown in bold in the following code. It needs to be added to the following methods and properties in the ArrayEx.vb file you created in Chapter 1 for Visual Basic users.

#### Visual Basic

```
Public Function Contains(ByVal item As T) As Boolean Implements ICollection(Of T).Contains  
Public Function Remove(ByVal item As T) As Boolean Implements ICollection(Of T).Remove  
Public Sub Add(ByVal item As T) Implements ICollection(Of T).Add  
Public Sub Clear() Implements ICollection(Of T).Clear  
Public ReadOnly Property Count() As Integer _  
    Implements ICollection(Of T).Count, ICollection.Count
```

## *CircularBuffer(T)*

Users cannot specifically state where they want to add an item or remove it, so only *ICollection* is implemented, and not *ICollection(T)*. First you need to add the *ICollection* interface to the *CircularBuffer(T)* class. Add a CircularBuffer.Collection.cs class file or a CircularBuffer.Collection.vb class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

#### C#

```
public class CircularBuffer
```

#### Visual Basic

```
Public Class CircularBuffer(Of T)
```

to this:

#### C#

```
public partial class CircularBuffer<T> : System.Collections.ICollection
```

#### Visual Basic

```
Partial Public Class CircularBuffer(Of T)  
    Implements ICollection
```

All of the methods and properties except *CopyTo*, *IsSynchronized*, and *SyncRoot* are already implemented.

The *IsSynchronized* and *SyncRoot* properties are discussed in detail in Chapter 8, so for now they are implemented as follows.

### C#

```
bool System.Collections.ICollection.IsSynchronized
{
    get { return false; }
}

object System.Collections.ICollection.SyncRoot
{
    get { throw new NotImplementedException(); }
}
```

### Visual Basic

```
Private ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized
    Get
        Return False
    End Get
End Property

Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property
```

Items have to be copied to the array from the start of the buffer to the end of the buffer. So you need to define a method as follows for copying.

### C#

```
void System.Collections.ICollection.CopyTo(Array array, int index)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (index < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }

    if (array.Rank > 1)
    {
        throw new RankException("Cannot handle multidimensional arrays");
    }

    if (array.Length - index < m_count)
    {
        throw new ArgumentException
            ("The specified array is not large enough to hold the data.", "array");
    }
}
```

```

for (int i = 0; i < m_count; ++i)
{
    array.SetValue(m_data[(m_start + i) % m_capacity], i + index);
}
}

```

### Visual Basic

```

Private Sub CopyTo(ByVal array As Array, ByVal index As Integer) _
    Implements ICollection.CopyTo
    If (array Is Nothing) Then
        Throw New ArgumentNullException("array")
    End If

    If (index < 0) Then
        Throw New ArgumentOutOfRangeException("arrayIndex")
    End If

    If (array.Rank > 1) Then
        Throw New RankException("Cannot handle multidimensional arrays")
    End If

    If (array.Length - index < m_count) Then
        Throw New ArgumentException(_
            ("array", "The specified array is not large enough to hold the data."))
    End If

    For i As Integer = 0 To m_count - 1
        array.SetValue(m_data((m_start + i) Mod m_capacity), i + index)
    Next
End Sub

```

The method starts at *m\_start* and counts up until it reaches the number of items in the buffer. This is accomplished by adding an index to *m\_start* and using the modulus operator by *m\_capacity*.

The *Implements* statement is shown in bold in the following code. It needs to be added to the following property in the CircularBuffer.vb file you created in Chapter 3 for Visual Basic users.

### Visual Basic

```
Public ReadOnly Property Count() As Integer Implements ICollection.Count
```

## *SingleLinkedList(T)* and *DoubleLinkedList(T)*

The implementation for the *SingleLinkedList(T)* and the *DoubleLinkedList(T)* classes is the same. First you need to add the *ICollection(T)* and *ICollection* interfaces to the *SingleLinkedList(T)* and the *DoubleLinkedList(T)* classes. Add a *SingleLinkedList.Collection.cs* class file and a *DoubleLinkedList.Collection.cs* class file to the C# project; or add the *SingleLinkedList.Collection.vb* class file or the *DoubleLinkedList.Collection.vb* class file to the Visual Basic project.

Change the class declaration for the *SingleLinkedList(T)* from the following:

**C#**

```
public class SingleLinkedList
```

**Visual Basic**

```
Public Class SingleLinkedList(Of T)
```

to this:

**C#**

```
public partial class SingleLinkedList<T> : ICollection<T>, System.Collection.ICollection
```

**Visual Basic**

```
Partial Public Class SingleLinkedList(Of T)
    Implements ICollection(Of T), ICollection
```

Change the class declaration for the *DoubleLinkedList(T)* from the following:

**C#**

```
public class DoubleLinkedList
```

**Visual Basic**

```
Public Class DoubleLinkedList(Of T)
```

to this:

**C#**

```
public partial class DoubleLinkedList <T> : ICollection<T>, System.Collection.ICollection
```

**Visual Basic**

```
Partial Public Class DoubleLinkedList(Of T)
    Implements ICollection(Of T), ICollection
```

All of the methods and properties except *Add*, *CopyTo*, *IsReadyOnly*, *IsSynchronized*, and *SyncRoot* are already implemented.

The *IsSynchronized* and *SyncRoot* properties are discussed in detail in Chapter 8, so for now they are implemented as follows.

**C#**

```
bool System.Collections.ICollection.IsSynchronized
{
    get { return false; }
}

object System.Collections.ICollection.SyncRoot
{
    get { throw new NotImplementedException(); }
}
```

**Visual Basic**

```

Private ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized
    Get
        Return False
    End Get
End Property

Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property

```

The *Add* method simply adds the specified item to the end of the linked list.

**C#**

```

void System.Collections.Generic.ICollection<T>.Add(T item)
{
    AddToEnd(item);
}

```

**Visual Basic**

```

Private Sub Add(ByVal item As T) Implements ICollection(Of T).Add
    AddToEnd(item)
End Sub

```

Both classes can be modified, so the *IsReadOnly* property needs to return *false*.

**C#**

```

/// <summary>
/// Gets a value indicating whether the collection is read only.
/// </summary>
public bool IsReadOnly
{
    get { return false; }
}

```

**Visual Basic**

```

''' <summary>
''' Gets a value indicating whether the collection is read only.
''' </summary>
Public ReadOnly Property IsReadOnly() As Boolean Implements ICollection(Of T).IsReadOnly
    Get
        Return False
    End Get
End Property

```

For the *CopyTo* method, you need to copy all nodes from the beginning of the list to the end of the list, as follows.

**C#**

```
/// <summary>
/// Copies the elements of the array to the specified array.
/// </summary>
/// <param name="array">The array to copy the data to.</param>
/// <param name="arrayIndex">The index in array where copying begins.</param>
public void CopyTo(T[] array, int arrayIndex)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (arrayIndex < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }

    if (array.Length - arrayIndex < m_count)
    {
        throw new ArgumentOutOfRangeException
            ("The specified array is not large enough to hold the data.", "array");
    }

    SingleLinkedListNode<T> current = Head;
    for (int i = 0; i < m_count; ++i)
    {
        array[i + arrayIndex] = current.Data;
        current = current.Next;
    }
}

void System.Collections.ICollection.CopyTo(Array array, int index)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (index < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }

    if (array.Rank > 1)
    {
        throw new RankException("Cannot handle multidimensional arrays");
    }
}
```

```
if (array.Length - index < m_count)
{
    throw new ArgumentException
        ("The specified array is not large enough to hold the data.", "array");
}

SingleLinkedListNode<T> current = Head;
for (int i = 0; i < m_count; ++i)
{
    array.SetValue(current.Data, i + index);
    current = current.Next;
}
}
```

### Visual Basic

```
'<summary>
' Copies the elements of the array to the specified array.
'</summary>
'<param name="array">The array to copy the data to.</param>
'<param name="arrayIndex">The index in array where copying begins.</param>
Public Sub CopyTo(ByVal array As T(), ByVal arrayIndex As Integer) _
    Implements ICollection(Of T).CopyTo
If (array Is Nothing) Then
    Throw New ArgumentNullException("array")
End If

If (arrayIndex < 0) Then
    Throw New ArgumentOutOfRangeException("arrayIndex")
End If

If (array.Length - arrayIndex < m_count) Then
    Throw New ArgumentOutOfRangeException _
        ("array", "The specified array is not large enough to hold the data.")
End If

Dim current As SingleLinkedListNode(Of T) = Head
For i As Integer = 0 To m_count - 1
    array(i + arrayIndex) = current.Data
    current = current.Next
Next
End Sub

Private Sub CopyTo1(ByVal array As Array, ByVal index As Integer) _
    Implements ICollection.CopyTo
If (array Is Nothing) Then
    Throw New ArgumentNullException("array")
End If

If (index < 0) Then
    Throw New ArgumentOutOfRangeException("arrayIndex")
End If

If (array.Rank > 1) Then
    Throw New RankException("Cannot handle multidimensional arrays")
End If
```

```

If (array.Length - index < m_count) Then
    Throw New ArgumentException _
        ("array", "The specified array is not large enough to hold the data.")
End If

Dim current As SingleLinkedListNode(Of T) = Head
For i As Integer = 0 To m_count - 1
    array.SetValue(current.Data, i + index)
    current = current.Next
Next
End Sub

```

The *CopyTo* method starts at *Head* and traverses the list until it gets to *Tail*, while keeping an index in variable *i*. The index is used to copy the current node's data to the index of (*i* + *index*) into the specified array. Don't forget to change the *SingleLinkedListNode(T)* to *DoubleLinkedListNode(T)* for the *DoubleLinkedList(T)* implementation.

The *Implements* statement is shown in bold in the following code. It needs to be added to the following methods and properties in the *SingleLinkedList.vb* and *DoubleLinkedList.vb* files you created in Chapter 1 for Visual Basic users.

#### Visual Basic

```

Public ReadOnly Property Count() As Integer _
    Implements ICollection.Count , ICollection(Of T).Count

Public Sub Clear() Implements ICollection(of T).Clear

Public Function Remove(ByVal item As T) As Boolean Implements ICollection(Of T).Remove

Public Function Contains(ByVal data As T) As Boolean Implements ICollection(Of T).Contains

```

## QueuedArray(*T*)

Users cannot specifically state where they want to add an item or remove it, so only *ICollection* is implemented, and not *ICollection(T)*. First you need to add the *ICollection* interface to the *QueuedArray(T)* class. Add a *QueuedArray.Collection.cs* class file or a *QueuedArray.Collection.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

C#

```
public class QueuedArray
```

#### Visual Basic

```
Public Class QueuedArray(Of T)
```

to this:

C#

```
public partial class QueuedArray<T> : System.Collections.ICollection
```

### Visual Basic

```
Partial Public Class QueuedArray(Of T)
    Implements ICollection
```

All of the methods and properties except *CopyTo*, *IsSynchronized*, and *SyncRoot* are already implemented.

The *IsSynchronized* and *SyncRoot* properties are discussed in detail in Chapter 8, so for now they are implemented as follows.

### C#

```
bool System.Collections.ICollection.IsSynchronized
{
    get { return false; }
}

object System.Collections.ICollection.SyncRoot
{
    get { throw new NotImplementedException(); }
}
```

### Visual Basic

```
Private ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized
    Get
        Return False
    End Get
End Property

Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property
```

Items need to be copied to the array from the start of the array to the end of the array. So you need to define a method as follows for copying.

### C#

```
void System.Collections.ICollection.CopyTo(Array array, int index)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (index < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }
```

```

if (array.Rank > 1)
{
    throw new RankException("Cannot handle multidimensional arrays");
}

if (array.Length - index < m_count)
{
    throw new ArgumentException
        ("array", "The specified array is not large enough to hold the data.");
}

for (int i = 0; i < m_count; ++i)
{
    array.SetValue(m_data[(m_head + i) % m_data.Length], i + index);
}
}

```

### Visual Basic

```

Private Sub CopyTo(ByVal array As Array, ByVal index As Integer) _
    Implements ICollection.CopyTo
If (array Is Nothing) Then
    Throw New ArgumentNullException("array")
End If

If (index < 0) Then
    Throw New ArgumentOutOfRangeException("arrayIndex")
End If

If (array.Rank > 1) Then
    Throw New RankException("Cannot handle multidimensional arrays")
End If

If (array.Length - index < m_count) Then
    Throw New ArgumentException _
        ("array", "The specified array is not large enough to hold the data.")
End If

For i As Integer = 0 To m_count - 1
    array.SetValue(m_data((m_head + i) Mod m_data.Length), i + index)
Next
End Sub

```

The method starts at *m\_head* and counts up until it reaches the number of items in the buffer. This is accomplished by adding an index to *m\_head* and then applying the modulus operator by *m\_data.Length*.

The *Implements* statement is shown in bold in the following code. It needs to be added to the following property in the *QueuedArray.vb* file you created in Chapter 3 for Visual Basic users.

### Visual Basic

```
Public ReadOnly Property Count() As Integer Implements ICollection.Count
```

## QueuedLinkedList(*T*)

Users cannot specifically state where they want to add an item or remove it, so only *ICollection* is implemented, and not *ICollection(T)*. First you need to add the *ICollection* interface to the *QueuedLinkedList(T)* class. Add a *QueuedLinkedList.Collection.cs* class file or a *QueuedLinkedList.Collection.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

**C#**

```
public class QueuedLinkedList
```

**Visual Basic**

```
Partial Public Class QueuedLinkedList(Of T)
```

to this:

**C#**

```
public partial class QueuedLinkedList<T> : System.Collections.ICollection
```

**Visual Basic**

```
Partial Public Class QueuedLinkedList(Of T)
    Implements ICollection
```

All of the methods and properties except *CopyTo*, *IsSynchronized*, and *SyncRoot* are already implemented.

The *IsSynchronized* and *SyncRoot* properties are discussed in detail in Chapter 8, so for now they are implemented as follows.

**C#**

```
bool System.Collections.ICollection.IsSynchronized
{
    get { return false; }
}

object System.Collections.ICollection.SyncRoot
{
    get { throw new NotImplementedException(); }
}
```

**Visual Basic**

```
Private ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized
    Get
        Return False
    End Get
End Property
```

```

Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property

```

Items are iterated in a queue in the same order they are iterated in the internal linked list data storage. So you can call the *CopyTo* method of the internal data storage instead.

#### C#

```

void System.Collections.ICollection.CopyTo(Array array, int index)
{
    ((System.Collections.ICollection)m_data).CopyTo(array, index);
}

```

#### Visual Basic

```

Private Sub CopyTo(ByVal array As Array, ByVal index As Integer) _
    Implements ICollection.CopyTo
    DirectCast(m_data, ICollection).CopyTo(array, index)
End Sub

```

The *Implements* statement is shown in bold in the following code. It needs to be added to the following property in the *QueuedLinkedList.vb* file you created in Chapter 3 for Visual Basic users.

#### Visual Basic

```
Public ReadOnly Property Count() As Integer Implements ICollection.Count
```

## *StackedArray(T)*

Users cannot specifically state where they want to add an item or remove it, so only *ICollection* is implemented and not *ICollection(T)*. First you need to add the *ICollection* interface to the *StackedArray(T)* class. Add a *StackedArray.Collection.cs* file or a *StackedArray.Collection.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

#### C#

```
public class StackedArray
```

#### Visual Basic

```
Public Class StackedArray(Of T)
```

to this:

#### C#

```
public partial class StackedArray<T> : System.Collections.ICollection
```

### Visual Basic

```
Partial Public Class StackedArray(Of T)
    Implements ICollection
```

All of the methods and properties except *CopyTo*, *IsSynchronized*, and *SyncRoot* are already implemented.

The *IsSynchronized* and *SyncRoot* properties are discussed in detail in Chapter 8, so for now they are implemented as follows.

### C#

```
bool System.Collections.ICollection.IsSynchronized
{
    get { return false; }
}

object System.Collections.ICollection.SyncRoot
{
    get { throw new NotImplementedException(); }
}
```

### Visual Basic

```
Private ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized
    Get
        Return False
    End Get
End Property

Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property
```

Stacks are implemented as LIFO, so items need to be copied to the specified array from the end of the internal storage to the beginning of the internal storage. So you need to define a method as follows for copying.

### C#

```
void System.Collections.ICollection.CopyTo(Array array, int index)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (index < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }
```

```

if (array.Rank > 1)
{
    throw new RankException("Cannot handle multidimensional arrays");
}

if (array.Length - index < m_data.Count)
{
    throw new ArgumentException
        ("array", "The specified array is not large enough to hold the data.");
}

for (int i = 0; i < m_data.Count; ++i)
{
    array.SetValue(m_data[m_data.Count - i - 1], i + index);
}
}

```

### Visual Basic

```

Private Sub CopyTo(ByVal array As Array, ByVal index As Integer) _
    Implements ICollection.CopyTo
If (array Is Nothing) Then
    Throw New ArgumentNullException("array")
End If

If (index < 0) Then
    Throw New ArgumentOutOfRangeException("arrayIndex")
End If

If (array.Rank > 1) Then
    Throw New RankException("Cannot handle multidimensional arrays")
End If

If (array.Length - index < m_data.Count) Then
    Throw New ArgumentException _
        ("array", "The specified array is not large enough to hold the data.")
End If

For i As Integer = 0 To m_data.Count - 1
    array.SetValue(m_data(m_data.Count - i - 1), i + index)
Next
End Sub

```

The method counts from *0* to *m\_data.Count* using variable *i*. It then takes the element at (*m\_data.Count – i – 1*) and assigns it to the specified array at (*index + i*).

The *Implements* statement is shown in bold in the following code. It needs to be added to the following property in the StackedArray.vb file you created in Chapter 3 for Visual Basic users.

### Visual Basic

```
Public ReadOnly Property Count() As Integer Implements ICollection.Count
```

## StackedLinkedList(*T*)

Users cannot specifically state where they want to add an item or remove it, so only *ICollection* is implemented and not *ICollection(T)*. First you need to add the *ICollection* interface to the *StackedLinkedList(T)* class. Add a *StackedLinkedList.Collection.cs* class file or a *StackedLinkedList.Collection.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

**C#**

```
public class StackedLinkedList
```

**Visual Basic**

```
Public Class StackedLinkedList(Of T)
```

to this:

**C#**

```
public partial class StackedLinkedList<T> : System.Collections.ICollection
```

**Visual Basic**

```
Partial Public Class StackedLinkedList(Of T)
    Implements ICollection
```

All of the methods and properties except *CopyTo*, *IsSynchronized*, and *SyncRoot* are already implemented.

The *IsSynchronized* and *SyncRoot* properties are discussed in detail in Chapter 8, so for now they are implemented as follows.

**C#**

```
bool System.Collections.ICollection.IsSynchronized
{
    get { return false; }
}

object System.Collections.ICollection.SyncRoot
{
    get { throw new NotImplementedException(); }
}
```

**Visual Basic**

```
Private ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized
    Get
        Return False
    End Get
End Property
```

```

Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property

```

Stacks are implemented as LIFO, so items need to be copied to the specified array from the end of the internal storage to the beginning of the internal storage. So you need to define a method as follows for copying.

### C#

```

void System.Collections.ICollection.CopyTo(Array array, int index)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (index < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }

    if (array.Rank > 1)
    {
        throw new RankException("Cannot handle multidimensional arrays");
    }

    if (array.Length - index < m_data.Count)
    {
        throw new ArgumentException
            ("array", "The specified array is not large enough to hold the data.");
    }

    DoubleLinkedListNode<T> current = m_data.Tail;
    for (int i = 0; i < m_data.Count; ++i)
    {
        array.SetValue(current.Data, i + index);
        current = current.Previous;
    }
}

```

### Visual Basic

```

Private Sub CopyTo(ByVal array As Array, ByVal index As Integer) _
    Implements ICollection.CopyTo
    If (array Is Nothing) Then
        Throw New ArgumentNullException("array")
    End If

    If (index < 0) Then
        Throw New ArgumentOutOfRangeException("arrayIndex")
    End If

```

```
If (array.Rank > 1) Then
    Throw New RankException("Cannot handle multidimensional arrays")
End If

If (array.Length - index < m_data.Count) Then
    Throw New ArgumentException _
        ("array", "The specified array is not large enough to hold the data.")
End If

Dim current As DoubleLinkedListNode(Of T) = m_data.Tail
For i As Integer = 0 To m_data.Count - 1
    array.SetValue(current.Data, i + index)
    current = current.Previous
Next
End Sub
```

The method traverses the list from the tail to the head while keeping track of the current index in *i*. The element at the current node then is copied to the index at (*i* + *index*) in the specified array.

The *Implements* statement is shown in bold in the following code. It needs to be added to the following property in the StackedLinkedList.vb file you created in Chapter 3 for Visual Basic users.

#### Visual Basic

```
Public ReadOnly Property Count() As Integer Implements ICollection.Count
```

## AssociativeArrayAL(*TKey, TValue*)

First you need to add the *ICollection* and *ICollection(T)* interfaces to the *AssociativeArrayAL(*TKey, TValue*)* class. Add an *AssociativeArrayAL.Collection.cs* class file or an *AssociativeArrayAL.Collection.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

#### C#

```
public class AssociativeArrayAL
```

#### Visual Basic

```
Public Class AssociativeArrayAL(Of TKey, TValue)
```

to this:

#### C#

```
public partial class AssociativeArrayAL<TKey, TValue> :
    System.Collections.ICollection,
    ICollection< KeyValuePair< TKey, TValue > >
```

**Visual Basic**

```
Partial Public Class AssociativeArrayAL(Of TKey, TValue)
    Implements ICollection, ICollection(Of KeyValuePair(Of TKey, TValue))
```

All of the methods and properties except *CopyTo*, *IsSynchronized*, and *SyncRoot* are already implemented.

The *IsSynchronized* and *SyncRoot* properties are discussed in detail in Chapter 8, so for now they are implemented as follows.

**C#**

```
bool System.Collections.ICollection.IsSynchronized
{
    get { return false; }
}

object System.Collections.ICollection.SyncRoot
{
    get { throw new NotImplementedException(); }
}
```

**Visual Basic**

```
Private ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized
    Get
        Return False
    End Get
End Property

Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property
```

The *AssociativeArrayAL(TKey,TValue)* can be modified, so the *IsReadOnly* property needs to return *false*.

**C#**

```
/// <summary>
/// Gets a value indicating whether the collection is read only.
/// </summary>
public bool IsReadOnly
{
    get { return false; }
}
```

**Visual Basic**

```
Public ReadOnly Property IsReadOnly() As Boolean _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).IsReadOnly
    Get
        Return False
    End Get
End Property
```

*T* for the *ICollection(T)* interface is defined as a *KeyValuePair(TKey,TValue)*. This allows you to traverse the collection and obtain the key/value pair at the same time. To do this, you need to convert from our internal structure to a *KeyValuePair(TKey,TValue)* structure. You also need to define an *Add(KeyValuePair(TKey,TValue))*, a *Contains(KeyValuePair(TKey,TValue))*, a *CopyTo*, and a *Remove(KeyValuePair(TKey,TValue))* method.

The *Add* method takes a *KeyValuePair(TKey,TValue)* as an argument. You need to pass the properties of the *Add* method parameter to the *Add(TKey key, TValue value)* method created in Chapter 2, as follows.

#### C#

```
void ICollection<KeyValuePair< TKey, TValue >>.Add(KeyValuePair< TKey, TValue > item)
{
    Add(item.Key, item.Value);
}
```

#### Visual Basic

```
Private Sub Add(ByVal item As KeyValuePair(Of TKey, TValue)) _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Add
    Add(item.Key, item.Value)
End Sub
```

The *Contains* method takes a *KeyValuePair(TKey,TValue)* as an argument.

#### C#

```
bool ICollection<KeyValuePair< TKey, TValue >>.Contains(KeyValuePair< TKey, TValue > item)
{
    EntryData data = FindKey(item.Key);
    if (data.IsEmpty)
    {
        return false;
    }

    return EqualityComparer< TValue >.Default.Equals(item.Value, data.Value);
}
```

#### Visual Basic

```
Private Function Contains(ByVal item As KeyValuePair(Of TKey, TValue)) As Boolean _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Contains
    Dim node As DoubleLinkedListNode(Of KVPair) = FindKey(item.Key)
    If (node Is Nothing) Then
        Return False
    End If

    Return EqualityComparer(Of TValue).Default.Equals(item.Value, node.Data.Value)
End Function
```

The *Contains* method uses the *FindKey* method to find the key. If the key is located, the *Contains* method then compares the value to verify that it's an exact match.

The *Remove* method takes a *KeyValuePair(TKey,TValue)* as an argument.

**C#**

```
bool ICollection<KeyValuePair<TKey, TValue>>.Remove(KeyValuePair<TKey, TValue> item)
{
    EntryData data = FindKey(item.Key);
    if (data.IsEmpty)
    {
        return false;
    }

    if (!EqualityComparer<TValue>.Default.Equals(item.Value, data.Value))
    {
        return false;
    }

    return Remove(data);
}
```

**Visual Basic**

```
Private Function Remove(ByVal item As KeyValuePair(Of TKey, TValue)) As Boolean _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Remove
    Dim node As DoubleLinkedListNode(Of KVPair) = FindKey(item.Key)
    If (node Is Nothing) Then
        Return False
    End If

    If (Not EqualityComparer(Of TValue).Default.Equals(item.Value, node.Data.Value)) Then
        Return False
    End If

    Return Remove(node)
End Function
```

The *Remove* method uses the *FindKey* method to find the key. If the key is located, the *Remove* method compares the value to verify that it's an exact match. If it is, the method then removes the item.

Both *CopyTo* methods traverse the collection and copy the elements to the specified array.

**C#**

```
void ICollection<KeyValuePair<TKey, TValue>>.CopyTo(KeyValuePair<TKey, TValue>[] array,
                                                    int arrayIndex)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (arrayIndex < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }

    if (array.Length - arrayIndex < Count)
    {
```

```
        throw new ArgumentOutOfRangeException
            ("array", "The specified array is not large enough to hold the data.");
    }

    int i = arrayIndex;
    foreach (KeyValuePair<TKey, TValue> kvp in this)
    {
        array[i] = kvp;
        ++i;
    }
}

void System.Collections.ICollection.CopyTo(Array array, int index)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (index < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }

    if (array.Rank > 1)
    {
        throw new RankException("Cannot handle multidimensional arrays");
    }

    if (array.Length - index < Count)
    {
        throw new ArgumentException
            ("array", "The specified array is not large enough to hold the data.");
    }

    int i = index;
    foreach (KeyValuePair<TKey, TValue> kvp in this)
    {
        array.SetValue(kvp, i);
        ++i;
    }
}
```

### Visual Basic

```
Private Sub CopyTo(ByVal array As KeyValuePair(Of TKey, TValue)(), _
                    ByVal arrayIndex As Integer) _
                    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).CopyTo
If (array Is Nothing) Then
    Throw New ArgumentNullException("array")
End If

If (arrayIndex < 0) Then
    Throw New ArgumentOutOfRangeException("arrayIndex")
End If

If (array.Length - arrayIndex < Count) Then
    Throw New ArgumentOutOfRangeException _
```

```

        ("array", "The specified array is not large enough to hold the data.")
End If

Dim i As Integer = arrayIndex
For Each kvp As KeyValuePair(Of TKey, TValue) In Me
    array(i) = kvp
    i += 1
Next
End Sub

Private Sub CopyTo1(ByVal array As Array, ByVal index As Integer) _
    Implements ICollection.CopyTo
If (array Is Nothing) Then
    Throw New ArgumentNullException("array")
End If

If (index < 0) Then
    Throw New ArgumentOutOfRangeException("arrayIndex")
End If

If (array.Rank > 1) Then
    Throw New RankException("Cannot handle multidimensional arrays")
End If

If (array.Length - index < Count) Then
    Throw New ArgumentException _
        ("array", "The specified array is not large enough to hold the data.")
End If

Dim i As Integer = index
For Each kvp As KeyValuePair(Of TKey, TValue) In Me
    array.SetValue(kvp, i)
    i += 1
Next
End Sub

```

The *Implements* statement is shown in bold in the following code. It needs to be added to the following methods and properties in the *AssociativeArrayAL.vb* file you created in Chapter 2 for Visual Basic users.

#### Visual Basic

```

Public ReadOnly Property Count() As Integer _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Count, ICollection.Count

Public Sub Clear() Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Clear

```

## *AssociativeArrayHT(TKey,TValue)*

First you need to add the *ICollection* and *ICollection(T)* interfaces to the *AssociativeArrayHT(TKey,TValue)* class. Add an *AssociativeArrayHT.Collection.cs* class file or an *AssociativeArrayHT.Collection.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

**C#**

```
public class AssociativeArrayHT
```

**Visual Basic**

```
Public Class AssociativeArrayHT(Of TKey, TValue)
```

to this:

**C#**

```
public partial class AssociativeArrayHT<TKey,TValue> :  
    System.Collection.ICollection,  
    ICollection< KeyValuePair<TKey,TValue> >
```

**Visual Basic**

```
Partial Public Class AssociativeArrayHT(Of TKey, TValue)  
    Implements ICollection, ICollection(Of KeyValuePair(Of TKey, TValue))
```

All of the methods and properties except *CopyTo*, *IsSynchronized*, and *SyncRoot* are already implemented.

The *IsSynchronized* and *SyncRoot* properties are discussed in detail in Chapter 8, so for now they are implemented as follows.

**C#**

```
bool System.Collections.ICollection.IsSynchronized  
{  
    get { return false; }  
}  
  
object System.Collections.ICollection.SyncRoot  
{  
    get { throw new NotImplementedException(); }  
}
```

**Visual Basic**

```
Private ReadOnly Property IsSynchronized() As Boolean Implements ICollection.IsSynchronized  
    Get  
        Return False  
    End Get  
End Property  
  
Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot  
    Get  
        Throw New NotImplementedException()  
    End Get  
End Property
```

*T* for the *ICollection(T)* interface is defined as a *KeyValuePair(TKey,TValue)*. This allows you to traverse the collection and obtain the key/value pair at the same time. To do this, you need to convert from our internal structure to a *KeyValuePair(TKey,TValue)* structure. You also need to define an *Add(KeyValuePair(TKey,TValue))*, a *Contains(KeyValuePair(TKey,TValue))*, a *CopyTo*, and a *Remove(KeyValuePair(TKey,TValue))* method.

The *Add* method takes a *KeyValuePair(TKey,TValue)* as an argument. You need to pass the properties of the *Add* method parameter to the *Add(TKey key, TValue value)* method created in Chapter 2, as follows.

#### C#

```
void ICollection<KeyValuePair< TKey, TValue >>.Add(KeyValuePair< TKey, TValue > item)
{
    Add(item.Key, item.Value);
}
```

#### Visual Basic

```
Private Sub Add(ByVal item As KeyValuePair(Of TKey, TValue)) _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Add
    Add(item.Key, item.Value)
End Sub
```

The *Contains* method takes a *KeyValuePair(TKey,TValue)* as an argument.

#### C#

```
bool ICollection<KeyValuePair< TKey, TValue >>.Contains(KeyValuePair< TKey, TValue > item)
{
    EntryData data = FindKey(item.Key);
    if (data.IsEmpty)
    {
        return false;
    }

    return EqualityComparer< TValue >.Default.Equals(item.Value, data.Value);
}
```

#### Visual Basic

```
Private Function Contains(ByVal item As KeyValuePair(Of TKey, TValue)) As Boolean _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Contains
    Dim data As EntryData = FindKey(item.Key)
    If (data.IsEmpty) Then
        Return False
    End If

    Return EqualityComparer(Of TValue).Default.Equals(item.Value, data.Value)
End Function
```

The *Contains* method uses the *FindKey* method to find the key. If the key is located, the *FindKey* method then compares the value to verify that it's an exact match.

The *Remove* method takes a *KeyValuePair(TKey,TValue)* as an argument.

**C#**

```
bool ICollection<KeyValuePair<TKey, TValue>>.Remove(KeyValuePair<TKey, TValue> item)
{
    EntryData data = FindKey(item.Key);
    if (data.IsEmpty)
    {
        return false;
    }

    if (!EqualityComparer<TValue>.Default.Equals(item.Value, data.Value))
    {
        return false;
    }

    return Remove(data);
}
```

**Visual Basic**

```
Private Function Remove(ByVal item As KeyValuePair(Of TKey, TValue)) As Boolean _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Remove
    Dim data As EntryData = FindKey(item.Key)
    If (data.IsEmpty) Then
        Return False
    End If

    If (Not EqualityComparer(Of TValue).Default.Equals(item.Value, data.Value)) Then
        Return False
    End If

    Return Remove(data)
End Function
```

The *Remove* method uses the *FindKey* method to find the key. If the key is located, the *FindKey* method then compares the value to verify that it's an exact match. If it is, the method then removes the item.

Both *CopyTo* methods traverse the collection and copy the elements to the specified array.

**C#**

```
void ICollection<KeyValuePair<TKey, TValue>>.CopyTo(KeyValuePair<TKey, TValue>[] array,
                                                       int arrayIndex)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (arrayIndex < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }
```

```

if (array.Length - arrayIndex < Count)
{
    throw new ArgumentOutOfRangeException
        ("array", "The specified array is not large enough to hold the data.");
}

int i = arrayIndex;
foreach (KeyValuePair<TKey, TValue> kvp in this)
{
    array[i] = kvp;
    ++i;
}
}

void System.Collections.ICollection.CopyTo(Array array, int index)
{
    if (array == null)
    {
        throw new ArgumentNullException("array");
    }

    if (index < 0)
    {
        throw new ArgumentOutOfRangeException("arrayIndex");
    }

    if (array.Rank > 1)
    {
        throw new RankException("Cannot handle multidimensional arrays");
    }

    if (array.Length - index < Count)
    {
        throw new ArgumentException
            ("array", "The specified array is not large enough to hold the data.");
    }

    int i = index;
    foreach (KeyValuePair<TKey, TValue> kvp in this)
    {
        array.SetValue(kvp, i);
        ++i;
    }
}

```

### Visual Basic

```

Private Sub CopyTo(ByVal array As KeyValuePair(Of TKey, TValue)(), _
                   ByVal arrayIndex As Integer) _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).CopyTo
If (array Is Nothing) Then
    Throw New ArgumentNullException("array")
End If

```

```
If (arrayIndex < 0) Then
    Throw New ArgumentOutOfRangeException("arrayIndex")
End If

If (array.Length - arrayIndex < Count) Then
    Throw New ArgumentOutOfRangeException _
        ("array", "The specified array is not large enough to hold the data.")
End If

Dim i As Integer = arrayIndex
For Each kvp As KeyValuePair(Of TKey, TValue) In Me
    array(i) = kvp
    i += 1
Next
End Sub

Private Sub CopyTo(ByVal array As Array, ByVal index As Integer) _
    Implements ICollection.CopyTo
    If (array Is Nothing) Then
        Throw New ArgumentNullException("array")
    End If

    If (index < 0) Then
        Throw New ArgumentOutOfRangeException("arrayIndex")
    End If

    If (array.Rank > 1) Then
        Throw New RankException("Cannot handle multidimensional arrays")
    End If

    If (array.Length - index < Count) Then
        Throw New ArgumentException _
            ("array", "The specified array is not large enough to hold the data.")
    End If

    Dim i As Integer = index
    For Each kvp As KeyValuePair(Of TKey, TValue) In Me
        array.SetValue(kvp, i)
        i += 1
    Next
End Sub
```

The *Implements* statement is shown in bold in the following code. It needs to be added to the following methods and properties in the *AssociativeArrayHT.vb* file created in Chapter 2 for Visual Basic users.

### Visual Basic

```
Public ReadOnly Property Count() As Integer _
    Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Count, ICollection.Count

Public Sub Clear() Implements ICollection(Of KeyValuePair(Of TKey, TValue)).Clear
```

## *IList* and *IList(T)* Overview

The *IList* and *IList(T)* interfaces define an object as a collection of objects that can be indexed individually. An indexed collection is a collection that can be accessed by an index in the same way that arrays are. The *IList* interface is used for generic and nongeneric indexed collections, such as the *Array*, *CollectionBase*, *ArrayList*, and *List(T)* classes. The *IList(T)* interface adds type-safe support and is used for generic collections such as *List(T)* and *SynchronizedCollection(T)*.

The following is the interface definition for the *IList(T)* interface.

### C#

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IComparable<T>
{
    T this[int index] { get; set; }
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);
}
```

### Visual Basic

```
Public Interface IList(Of T)
    Inherits ICollection(Of T), IEnumerable(Of T), IComparable(Of T)
    Function IndexOf(ByVal item As T) As Integer
    Sub Insert(ByVal index As Integer, ByVal item As T)
    Sub RemoveAt(ByVal index As Integer)
    Default Property Item(ByVal index As Integer) As T
End Interface
```

Here is the interface definition for the *IList* interface.

### C#

```
public interface IList : ICollection, IEnumerable
{
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[int index] { get; set; }
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);
}
```

### Visual Basic

```
Public Interface IList
    Inherits ICollection, IEnumerable
    Function Add(ByVal value As Object) As Integer
    Sub Clear()
    Function Contains(ByVal value As Object) As Boolean
    Function IndexOf(ByVal value As Object) As Integer
    Sub Insert(ByVal index As Integer, ByVal value As Object)
    Sub Remove(ByVal value As Object)
    Sub RemoveAt(ByVal index As Integer)
    ReadOnly Property IsFixedSize As Boolean
    ReadOnly Property IsReadOnly As Boolean
    Default Property Item(ByVal index As Integer) As Object
End Interface
```

At first glance, it looks as if the *IList* interface has more functionality than the *IList(T)*. The *IList(T)* interface derives from the *ICollection(T)* interface, which contains all of the methods and properties in *IList* except *IsFixedSize*, but in type-safe form. *IList* contains the nongeneric form of *Add*, *Item*, *Contains*, *IndexOf*, *Insert*, and *Remove*, and the *IList(T)* interface contains the generic form of these methods and properties. Through these two interfaces, you are able to add, insert, remove, and access objects in an indexed type collection.

The *IsFixedSize* property in *IList* is used to state whether the collection has a fixed size. A collection that is fixed size can not grow or shrink, so any method's calls, such as *Clear*, *Add*, *Remove*, *RemoveAt*, or *Insert*, are an invalid operation. However, you can set the elements in the collection to a different value.

In addition to the *Add* method being a nongeneric type in *IList* and a generic type in *IList(T)*, the *IList.Add* method returns an integer. The return value is the index of where the item was added.

## Adding *IList(T)* and *IList* Support to Classes

Adding the *IList(T)* and *IList* to your class allows others to access the elements in your class through indexing. Users are also able to add, remove, index, and check the contents of your class in the same way they do with the *List(T)* and other classes that implement the interface.

### *ArrayEx(T)*

First you need to add the *IList(T)* and *IList* interfaces to the *ArrayEx(T)* class. Add an *ArrayEx.List.cs* class file or an *ArrayEx.List.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

**C#**

```
public class ArrayEx
```

**Visual Basic**

```
Public Class ArrayEx(Of T)
```

to this:

**C#**

```
public partial class ArrayEx<T> : IList<T>, System.Collections.IList
```

**Visual Basic**

```
Partial Public Class ArrayEx(Of T)
    Implements IList(Of T), IList
```

All of the methods and properties except *IsFixedSize* and the non-generic versions of *Add*, *Contains*, *IndexOf*, *Insert*, *Remove*, and *Item* are already implemented.

Because *ArrayEx(T)* can be resized, you need to return a *false* for the *IsFixedSize* property, as follows.

**C#**

```
bool System.Collections.IList.IsFixedSize
{
    get { return false; }
}
```

**Visual Basic**

```
Private ReadOnly Property IsFixedSize() As Boolean Implements IList.IsFixedSize
    Get
        Return False
    End Get
End Property
```

You need to return a *true* if you want the collection to be of a fixed size after it is created.

You also need to implement the nongeneric versions of *Add*, *Contains*, *IndexOf*, *Insert*, *Remove*, and *Item*. For these methods and properties, you can cast the specified object in the parameter to type *T* and then call the generic equivalent of the method or property you are in so that you do not have to duplicate any code. To do this, you need to check and see if the specified object can be cast to type *T* in each one of the methods and properties. The easiest way to do this is to create a method that does the check for you and call it at the beginning of each of the methods and properties. You can use the following method to do so.

**C#**

```
bool CanCastToT(object value)
{
    if (!(value is T) && ((value != null) || typeof(T).IsValueType))
    {
        return false;
    }
    return true;
}
```

**Visual Basic**

```
Private Function CanCastToT(ByVal value As Object)
    If (Not (TypeOf value Is T) And ((value IsNot Nothing) Or GetType(T).IsValueType)) Then
        Return False
    End If
    Return True
End Function
```

The *CanCastToT* method verifies that the specified object can be cast to type *T*. The method first checks to see whether the value is of type *T* by doing *!(value is T)* (using C#). This check fails if *value* is not of type *T* or it is *null*. If it fails, you need to ensure that it is not because *value* is *null*. If *value* is *null*, you need to make sure *T* is not a value type, because *null* values cannot be assigned to a value type. You can check for that by doing an *AND* operation with *((value != null) || typeof(T).IsValueType)* (using C#). The *typeof(T).IsValueType* property returns *true* if type *T* is a value type.

Now that you have created a method to check whether the specified object can be cast to *T*, you can implement the remaining methods and properties.

**C#**

```
bool System.Collections.IList.Contains(object value)
{
    if (CanCastToT(value))
    {
        throw new ArgumentException("Value is not of type T", "value");
    }
    return Contains((T)value);
}

int System.Collections.IList.IndexOf(object value)
{
    if (CanCastToT(value))
    {
        throw new ArgumentException("Value is not of type T", "value");
    }
    return IndexOf((T)value);
}
```

```

void System.Collections.IList.Insert(int index, object value)
{
    if (CanCastToT(value))
    {
        throw new ArgumentException("Value is not of type T", "value");
    }
    Insert(index, (T)value);
}

void System.Collections.IList.Remove(object value)
{
    if (CanCastToT(value))
    {
        throw new ArgumentException("Value is not of type T", "value");
    }
    Remove((T)value);
}

object System.Collections.IList.this[int index]
{
    get
    {
        return this[index];
    }
    set
    {
        if (CanCastToT(value))
        {
            throw new ArgumentException("Value is not of type T", "value");
        }
        this[index] = (T)value;
    }
}

int System.Collections.IList.Add(object value)
{
    if (CanCastToT(value))
    {
        throw new ArgumentException("Value is not of type T", "value");
    }

    Add((T)value);

    return Count - 1;
}

```

### Visual Basic

```

Private Function Add(ByVal value As Object) As Integer Implements IList.Add
    If (CanCastToT(value)) Then
        Throw New ArgumentException("Value is not of type T", "value")
    End If

```

```
    Add(DirectCast(value, T))

    Return Count - 1
End Function

Private Function Contains(ByVal value As Object) As Boolean Implements IList.Contains
    If (CanCastToT(value)) Then
        Throw New ArgumentException("Value is not of type T", "value")
    End If
    Return Contains(DirectCast(value, T))
End Function

Private Function IndexOf(ByVal value As Object) As Integer Implements IList.IndexOf
    If (CanCastToT(value)) Then
        Throw New ArgumentException("Value is not of type T", "value")
    End If
    Return IndexOf(DirectCast(value, T))
End Function

Private Sub Insert(ByVal index As Integer, ByVal value As Object) Implements IList.Insert
    If (CanCastToT(value)) Then
        Throw New ArgumentException("Value is not of type T", "value")
    End If
    Insert(index, DirectCast(value, T))
End Sub

Private Sub Remove(ByVal value As Object) Implements IList.Remove
    If (CanCastToT(value)) Then
        Throw New ArgumentException("Value is not of type T", "value")
    End If
    Remove(DirectCast(value, T))
End Sub

Private Property Item2(ByVal index As Integer) As Object Implements IList.Item
    Get
        Return Me(index)
    End Get
    Set(ByVal value)
        If (CanCastToT(value)) Then
            Throw New ArgumentException("Value is not of type T", "value")
        End If
        Me(index) = DirectCast(value, T)
    End Set
End Property
```

Each method and property calls the *CanCastToT* method to verify that *value* can be cast to type *T*. If the specified object cannot be cast to *T*, an *ArgumentException* is thrown to notify the caller that he or she specified an invalid argument type. Also, do you remember that earlier you learned that the return type of the *Add* method of *IList* is different than the *IList(T)*? The preceding implementation for the *Add* method returns a (*Count - 1*), which is the location of the item after the *ArrayEx(T).Add* method executes.

The *Implements* statement is shown in bold in the following code. It needs to be added to the following methods and properties in the ArrayEx.vb file created in Chapter 1 for Visual Basic users.

### Visual Basic

```
Default Public Property Item(ByVal index As Integer) As T Implements IList(Of T).Item

Public Function IndexOf(ByVal item As T) As Integer Implements IList(Of T).IndexOf

Public Sub Insert(ByVal index As Integer, ByVal item As T) Implements IList(Of T).Insert

Public Sub Clear() Implements ICollection(Of T).Clear, IList.Clear

Public Sub RemoveAt(ByVal index As Integer) Implements IList(Of T).RemoveAt, IList.RemoveAt
```

## *IDictionary(TKey,TValue)* Overview

The *IDictionary* and *IDictionary(TKey,TValue)* interfaces define an object as a collection of key/value pairs.

### C#

```
public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>,  
IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable  
{  
    // Methods  
    void Add(TKey key, TValue value);  
    bool ContainsKey(TKey key);  
    bool Remove(TKey key);  
    bool TryGetValue(TKey key, out TValue value);  
  
    // Properties  
    TValue this[TKey key] { get; set; }  
    ICollection<TKey> Keys { get; }  
    ICollection<TValue> Values { get; }  
}  
public interface IDictionary : ICollection, IEnumerable  
{  
    // Methods  
    void Add(object key, object value);  
    void Clear();  
    bool Contains(object key);  
    IDictionaryEnumerator Getenumerator();  
    void Remove(object key);  
  
    // Properties  
    bool IsFixedSize { get; }  
    bool IsReadOnly { get; }  
    object this[object key] { get; set; }  
    ICollection Keys { get; }  
    ICollection Values { get; }  
}
```

### Visual Basic

```
Public Interface IDictionary(Of TKey, TValue)
    Inherits ICollection(Of KeyValuePair(Of TKey, TValue)), _
        IEnumerable(Of KeyValuePair(Of TKey, TValue)), IEnumerable
    ' Methods
    Sub Add(ByVal key As TKey, ByVal value As TValue)
    Function ContainsKey(ByVal key As TKey) As Boolean
    Function Remove(ByVal key As TKey) As Boolean
    Function TryGetValue(ByVal key As TKey, <Out> ByRef value As TValue) As Boolean

    ' Properties
    Default Property Item(ByVal key As TKey) As TValue
    ReadOnly Property Keys As ICollection(Of TKey)
    ReadOnly Property Values As ICollection(Of TValue)
End Interface

Public Interface IDictionary
    Inherits ICollection, IEnumerable
    ' Methods
    Sub Add(ByVal key As Object, ByVal value As Object)
    Sub Clear()
    Function Contains(ByVal key As Object) As Boolean
    Function GetEnumerator() As IDictionaryEnumerator
    Sub Remove(ByVal key As Object)

    ' Properties
    ReadOnly Property IsFixedSize As Boolean
    ReadOnly Property IsReadOnly As Boolean
    Default Property Item(ByVal key As Object) As Object
    ReadOnly Property Keys As ICollection
    ReadOnly Property Values As ICollection
End Interface
```

The *IDictionary* and *IDictionary(TKey, TValue)* interfaces both allow you to enumerate the object's key and value as a pair. They do this by returning a key/value pair object called *DictionaryEntry* and *KeyValuePair(TKey, TValue)* respectively. Both objects contain the value and the key that goes with the value. In Chapter 2, you learned that a dictionary is another name for an associative array, so you can also use the interfaces to access associative arrays.

## Adding Key/Value Pair Support to Classes

Adding the *IDictionary(TKey, TValue)* to your class allows others to access your class as a dictionary (also called an *associative array*). Users are able to traverse keys and values. They can also associate keys to values and retrieve values by using keys.

## **AssociativeArrayAL(TKey,TValue)**

First you need to add the *IDictionary(TKey,TValue)* interface to the *AssociativeArrayAL(TKey,TValue)* class. Add an *AssociativeArrayAL.Dictionary.cs* class file or an *AssociativeArrayAL.Dictionary.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

### C#

```
public class AssociativeArrayAL
```

### Visual Basic

```
Public Class AssociativeArrayAL(Of TKey, TValue)
```

to this:

### C#

```
public partial class AssociativeArrayAL<TKey, TValue> : IDictionary<TKey, TValue>
```

### Visual Basic

```
Partial Public Class AssociativeArrayAL(Of TKey, TValue)
    Implements IDictionary(Of TKey, TValue)
```

The *IDictionary(TKey,TValue)* interface defines the *Keys* and *Values* properties, so you need to remove the old implementation you created in Chapter 2. The new implementation returns an *ICollection(TKey)* and *ICollection(TValue)* respectively.

### C#

```
public ICollection<TKey> Keys
{
    get { return new KeyCollection(this); }
}

public ICollection<TValue> Values
{
    get { return new ValueCollection(this); }
}
```

### Visual Basic

```
Public ReadOnly Property Keys() As ICollection(Of TKey) _
    Implements IDictionary(Of TKey, TValue).Keys
```

Get

```
    Return New KeyCollection(Me)
```

End Get

End Property

```
Public ReadOnly Property Values() As ICollection(Of TValue) _
    Implements IDictionary(Of TKey, TValue).Values
```

Get

```
    Return New ValueCollection(Me)
```

End Get

End Property

You need to create a *KeyCollection* and *ValueCollection* class for the *Keys* and *Values* properties. The full implementation for the *KeyCollection* and *ValueCollection* properties can be found in the *AssociativeArrayAL(TKey, TValue)* partial class located in the Chapter 6\CS \DevGuideToCollections\AssociativeArrayAL.Dictionary.cs file for C# or the Chapter 6\VB \DevGuideToCollections\AssociativeArrayAL.Dictionary.vb file for Visual Basic.



**More Info** See the section titled “Adding Enumeration Support to Classes” if you have questions about the implementation.

The *Implements* statement is shown in bold in the following code. It needs to be added to the following methods and properties in the *AssociativeArrayAL.vb* file you created in Chapter 2 for Visual Basic users.

#### Visual Basic

```
Public Function ContainsKey(ByVal key As TKey) As Boolean _
    Implements IDictionary(Of TKey, TValue).ContainsKey

Public Function Remove(ByVal key As TKey) As Boolean _
    Implements IDictionary(Of TKey, TValue).Remove

Default Public Property Item(ByVal key As TKey) As TValue _
    Implements IDictionary(Of TKey, TValue).Item

Public Function TryGetValue(ByVal key As TKey, ByRef value As TValue) As Boolean _
    Implements IDictionary(Of TKey, TValue).TryGetValue

Public Sub Add(ByVal key As TKey, ByVal value As TValue) _
    Implements IDictionary(Of TKey, TValue).Add
```

## AssociativeArrayHT(TKey,TValue)

First you need to add the *IDictionary(TKey,TValue)* interface to the *AssociativeArrayHT-(TKey,TValue)* class. Add an *AssociativeArrayHT.Dictionary.cs* class file or an *AssociativeArrayHT.Dictionary.vb* class file to the C# or Visual Basic project you worked with in Chapters 1 through 3. Change the class declaration from the following:

C#

```
public class AssociativeArrayHT
```

to this:

C#

```
public partial class AssociativeArrayHT<TKey, TValue> : IDictionary<TKey, TValue>
```

The *IDictionary(TKey,TValue)* interface defines the *Keys* and *Values*, so you need to remove the old implementation you created in Chapter 2. The new implementation returns an *ICollection(TKey)* and *ICollection(TValue)* respectively.

#### C#

```
public ICollection<TKey> Keys
{
    get { return new KeyCollection(this); }
}

public ICollection<TValue> Values
{
    get { return new ValueCollection(this); }
}
```

#### Visual Basic

```
Public ReadOnly Property Keys() As ICollection(Of TKey) _
    Implements IDictionary(Of TKey, TValue).Keys
Get
    Return New KeyCollection(Me)
End Get
End Property

Public ReadOnly Property Values() As ICollection(Of TValue) _
    Implements IDictionary(Of TKey, TValue).Values
Get
    Return New ValueCollection(Me)
End Get
End Property
```

You need to create a *KeyCollection* and *ValueCollection* class for the *Keys* and *Values* properties. The full implementation for the *KeyCollection* and *ValueCollection* classes can be found in the *AssociativeArrayHT(TKey, TValue)* partial class located in the Chapter 6\CS\DevGuideToCollections\AssociativeArrayHT.Dictionary.cs file for C# or the Chapter 6\VB\DevGuideToCollections\AssociativeArrayHT.Dictionary.vb file for Visual Basic.



**More Info** See the section titled “Adding Enumeration Support to Classes” if you have questions about the implementation.

The *Implements* statement is shown in bold in the following code. It needs to be added to the following methods and properties in the *AssociativeArrayHT.vb* file you created in Chapter 2 for Visual Basic users.

### Visual Basic

```
Public Function ContainsKey(ByVal key As TKey) As Boolean _
    Implements IDictionary(Of TKey, TValue).ContainsKey

Public Function Remove(ByVal key As TKey) As Boolean _
    Implements IDictionary(Of TKey, TValue).Remove

Default Public Property Item(ByVal key As TKey) As TValue _
    Implements IDictionary(Of TKey, TValue).Item

Public Function TryGetValue(ByVal key As TKey, ByRef value As TValue) As Boolean _
    Implements IDictionary(Of TKey, TValue).TryGetValue

Public Sub Add(ByVal key As TKey, ByVal value As TValue) _
    Implements IDictionary(Of TKey, TValue).Add
```

## Summary

In this chapter, you saw how to implement the common .NET collection interfaces for your classes. You saw that the *IEnumerable(T)* and *IEnumerable(T)* interfaces allow your classes to be enumerated. You also saw that the *ICollection* interface allows your collection to return the length of the collection and to support synchronization. Implementing the *ICollection(T)* interface gives your collection common collection support. You also saw how the *ICollection(T)* and *ICollection* interfaces are not derived from each other. You then saw how to add indexing support to your collections through the *IList(T)* interface and how to add dictionary support through the *IDictionary(TKey, TValue)* interface. All of the interfaces discussed in this chapter allow your collections to be used like any other .NET collections, as Chapter 7 partially demonstrates.



# Chapter 7

# Introduction to LINQ

After completing this chapter, you will be able to

- Understand what LINQ is.
- Write LINQ queries against collection classes.

## What Is LINQ?

In the previous chapters, you saw how to create and interact with collections using the `foreach` statement and Microsoft .NET interfaces. In this chapter, you will discover a different approach: how to query your collections with the Language Integrated Query (LINQ). With LINQ, you can query data by using native language syntax.

Here's an example that does not use LINQ.

### C#

```
Random rnd = new Random();
List<int> randomNumbers = new List<int>();
for (int i = 0; i < 10; i++)
{
    randomNumbers.Add(rnd.Next(100));
}

List<int> sortedNumbers = new List<int>(randomNumbers);

sortedNumbers.Sort();
sortedNumbers.Reverse();
```

### Visual Basic

```
Dim rnd As Random = New Random()
Dim randomNumbers As New List(Of Integer)()
For i As Integer = 0 To 9
    randomNumbers.Add(rnd.Next(100))
Next

Dim sortedNumbers As New List(Of Integer)(randomNumbers)

sortedNumbers.Sort()
sortedNumbers.Reverse()
```

And here's an example that uses LINQ.

#### C#

```
Random rnd = new Random();
var randomNumbers = from n in Enumerable.Range(1, 10) select rnd.Next(100);
var sortedNumbers = from random in randomNumbers orderby random descending select random;
```

#### Visual Basic

```
Dim rnd As Random = New Random()
Dim randomNumbers = From n In Enumerable.Range(1, 10) Select rnd.Next(100)
Dim sortedNumbers = From random In randomNumbers Order By random Descending Select random
```

Both examples create a random list of numbers sorted in descending order. Notice how the LINQ code did not require the *Sort* and *Reverse* methods. Using LINQ, you can also nest the statements just shown as follows.

#### C#

```
Random rnd = new Random();

var sortedNumbers = from random in (from n in Enumerable.Range(1, 10) select
rnd.Next(100)) orderby random descending select random;
```

#### Visual Basic

```
Dim rnd As Random = New Random()

Dim sortedNumbers = From random In (From n In Enumerable.Range(1, 10) Select _
rnd.Next(100)) Order By random Descending Select random
```

## LINQ Basics

To get started writing LINQ queries, it helps to know the three distinct actions of a LINQ query as defined in MSDN.

1. Obtain the data source.
2. Create the query.
3. Execute the query.

## Potential LINQ Data Sources

As defined on MSDN, at <http://msdn.microsoft.com/en-us/library/bb397906.aspx>, the data sources that you can use with LINQ are

*XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.*

You can use .NET collections—the only data source that this chapter focuses on—as data sources because they all implement the *IEnumerable* or *IEnumerable(T)* interfaces. Because the classes you created in Chapter 6, “.NET Collection Interfaces,” also implement *IEnumerable(T)*, you can use them in a LINQ query such as the following.

### C#

```
ArrayEx<int> array = new ArrayEx<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 });

var oddInts = from element in array where (element % 2) == 1 select element;

foreach (int odd in oddInts)
{
    Console.WriteLine(odd);
}
```

### Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim oddInts = From element In array Where (element Mod 2) = 1 Select element

For Each odd As Integer In oddInts
    Console.WriteLine(odd)
Next
```

### Output

```
1
3
5
7
9
```

The preceding code executes a query that returns all odd numbers from the variable *array* and then writes them to the console.

## What You Should Know About Query Execution

It is important to know that the actual query does not execute until you iterate over the object returned by the query.

### C#

```
ArrayEx<int> array = new ArrayEx<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 });

Console.WriteLine("Building the query");
var oddInts = from element in array where (element % 2) == 1 select element;

Console.WriteLine("Adding 11, 12, and 13 to the data source");
array.Add(11);
array.Add(12);
array.Add(13);
```

```
Console.WriteLine("Iterating over the query");

foreach (int odd in oddInts)
{
    Console.WriteLine(odd);
}
```

### Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Console.WriteLine("Building the query")
Dim oddInts = From element In array Where (element Mod 2) = 1 Select element

Console.WriteLine("Adding 11, 12, and 13 to the data source")
array.Add(11)
array.Add(12)
array.Add(13)

Console.WriteLine("Iterating over the query")

For Each odd As Integer In oddInts
    Console.WriteLine(odd)
Next
```

### Output

```
Building the query
Adding 11, 12, and 13 to the data source
Iterating over the query
1
3
5
7
9
11
13
```

The preceding example shows how you can easily introduce a bug into your application. Queries execute when they are iterated over—not when they are created. The query itself creates an “execution plan.” That plan was stored in *oddInts* but not executed until the code iterated over *oddInts*. Note that during the time between the creation and execution of the query, the numbers 11, 12, and 13 were added to the data source *array* and showed up in the results.

Also note that the query executes *every time* you iterate over the query variable, not only the first time, as the next example shows you.

### C#

```
ArrayEx<int> array = new ArrayEx<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 });

Console.WriteLine("Building the query");
```

```
var oddInts = from element in array where (element % 2) == 1 select element;

Console.WriteLine("Adding 11, 12, and 13 to the data source");
array.Add(11);
array.Add(12);
array.Add(13);

Console.WriteLine("Iterating over the query");

foreach (int odd in oddInts)
{
    Console.WriteLine(odd);
}

Console.WriteLine("Adding 14, 15, 16, and 17 to the data source");
array.Add(14);
array.Add(15);
array.Add(16);
array.Add(17);

Console.WriteLine("Iterating over the query");
foreach (int odd in oddInts)
{
    Console.WriteLine(odd);
}
```

### Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Console.WriteLine("Building the query")
Dim oddInts = From element In array Where (element Mod 2) = 1 Select element

Console.WriteLine("Adding 11, 12, and 13 to the data source")
array.Add(11)
array.Add(12)
array.Add(13)

Console.WriteLine("Iterating over the query")

For Each odd As Integer In oddInts
    Console.WriteLine(odd)
Next

Console.WriteLine("Adding 14, 15, 16, and 17 to the data source")
array.Add(14)
array.Add(15)
array.Add(16)
array.Add(17)

Console.WriteLine("Iterating over the query")
For Each odd As Integer In oddInts
    Console.WriteLine(odd)
Next
```

### Output

```
Building the query
Adding 11, 12, and 13 to the data source
Iterating over the query
1
3
5
7
9
11
13
Adding 14, 15, 16, and 17 to the data source
Iterating over the query
1
3
5
7
9
11
13
15
17
```

The two iterations over the `oddInts` variable provided two different results because items were added to the original data source before each iteration. This is because the query variable holds the query command (the execution plan) and not the query result.

## Forcing Immediate Query Execution

You may sometimes find it useful to execute a query immediately for performance reasons, code readability, or to retrieve the data before the collection is modified.

The `ToList` and `ToArray` methods force immediate execution of the query, as shown in the following example.

### C#

```
ArrayEx<int> array = new ArrayEx<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 });

Console.WriteLine("Building the query");
var oddInts = (from element in array where (element % 2) == 1 select element).ToArray();

Console.WriteLine("Adding 11, 12, and 13 to the data source");
array.Add(11);
array.Add(12);
array.Add(13);

Console.WriteLine("Iterating over the query");

foreach (int odd in oddInts)
{
    Console.WriteLine(odd);
}
```

```
Console.WriteLine("Adding 14, 15, 16, and 17 to the data source");
array.Add(14);
array.Add(15);
array.Add(16);
array.Add(17);

Console.WriteLine("Iterating over the query");
foreach (int odd in oddInts)
{
    Console.WriteLine(odd);
}
```

### Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Console.WriteLine("Building the query")
Dim oddInts = (From element In array Where (element Mod 2) = 1 Select element).ToArray()

Console.WriteLine("Adding 11, 12, and 13 to the data source")
array.Add(11)
array.Add(12)
array.Add(13)

Console.WriteLine("Iterating over the query")

For Each odd As Integer In oddInts
    Console.WriteLine(odd)
Next

Console.WriteLine("Adding 14, 15, 16, and 17 to the data source")
array.Add(14)
array.Add(15)
array.Add(16)
array.Add(17)

Console.WriteLine("Iterating over the query")
For Each odd As Integer In oddInts
    Console.WriteLine(odd)
Next
```

### Output

```
Building the query
Adding 11, 12, and 13 to the data source
Iterating over the query
1
3
5
7
9
```

Adding 14, 15, 16, and 17 to the data source

Iterating over the query

1  
3  
5  
7  
9

The *oddInts* variable now stores the results of the query rather than the query commands. The *ToArray(T)* method returns a *T[]* and the *ToList(T)* method returns a *List(T)*.

Methods that perform aggregation functions over the query, such as *Count*, *Max*, *Average*, and *First*, also execute the query because they must perform a *foreach* to obtain their result. These methods return a single value and not an enumeration.

#### C#

```
ArrayEx<int> array = new ArrayEx<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 });

Console.WriteLine("Building the query");
var count = (from element in array where (element % 2) == 1 select element).Count();

Console.WriteLine("Adding 11, 12, and 13 to the data source");
array.Add(11);
array.Add(12);
array.Add(13);

Console.WriteLine("Count of the original query is still {0}", count);

Console.WriteLine("Adding 14, 15, 16, and 17 to the data source");
array.Add(14);
array.Add(15);
array.Add(16);
array.Add(17);

Console.WriteLine("Count of the original query is still {0}", count);
```

#### Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Console.WriteLine("Building the query")
Dim count = (From element In array Where (element Mod 2) = 1 Select element).Count()

Console.WriteLine("Adding 11, 12, and 13 to the data source")
array.Add(11)
array.Add(12)
array.Add(13)
```

```
Console.WriteLine("Count of the original query is still {0}", count)

Console.WriteLine("Adding 14, 15, 16, and 17 to the data source")
array.Add(14)
array.Add(15)
array.Add(16)
array.Add(17)

Console.WriteLine("Count of the original query is still {0}", count)
```

### Output

```
Building the query
Adding 11, 12, and 13 to the data source
Count of the original query is still 5
Adding 14, 15, 16, and 17 to the data source
Count of the original query is still 5
```

The *count* variable never changes because the query was executed when it was created.

## Getting Started with LINQ

This chapter is designed to quickly get you up and running so that you can use LINQ to interact with collections. Although you can use the examples in this chapter with other data sources, it is not the intention of this book to be a complete guide on LINQ. Many books have been written about LINQ, and I would suggest purchasing one or more of them to delve deeper into LINQ after reading this chapter. From this point on, the examples in this book will mainly use LINQ—except when doing so is not possible or when using something else makes the code easier to understand. The examples in this chapter use the following structure.

### C#

```
public enum Sex
{
    Unknown,
    Male,
    Female,
}

public struct Person
{
    public string FirstName;
    public string LastName;
    public int Age;
    public Sex Sex;
```

```

public Person(string first, string last)
{
    FirstName = first;
    LastName = last;

    // Placeholder for examples that do not need an age or sex.
    Age = 18;
    Sex = Sex.Unknown;
}

public Person(string first, string last, int age, Sex sex)
{
    FirstName = first;
    LastName = last;
    Age = age;
    Sex = sex;
}
}

```

### Visual Basic

```

Public Enum Sex
    Unknown
    Male
    Female
End Enum

Public Structure Person
    Public FirstName As String
    Public LastName As String
    Public Age As Integer
    Public Sex As Sex

    Public Sub New(ByVal first As String, ByVal last As String)
        FirstName = first
        LastName = last

        ' Placeholder for examples that do not need an age or sex.
        Age = 18
        Sex = Sex.Unknown
    End Sub

    Public Sub New(ByVal first As String, ByVal last As String, ByVal age As Integer, _
                  ByVal sex As Sex)
        Me.FirstName = first
        Me.LastName = last
        Me.age = age
        Me.sex = sex
    End Sub
End Structure

```

Using this structure in the following examples will help with some of the complex examples.

## Additions to the .NET Language for LINQ

The following sections contain information about changes to the programming languages that were made for LINQ. You may find some of these useful when programming with LINQ and also when doing general programming.

### The `var` Keyword

The previous Microsoft Visual C# examples in this chapter contained a new keyword called `var`, used to define local variables. The `var` keyword tells the compiler to infer the data type of the variable from the expression on the right side. For example, the following example infers the data type of the variable `v1` as `bool`. In Microsoft Visual Basic, you don't need the `var` keyword; you simply leave out the "As <Type>" portion of the variable definition.

#### C#

```
var v1 = 0 == 1;
```

#### Visual Basic

```
Dim v1 = 0 = 1
```

There are several things you should know about this new keyword.

First, the variable must be initialized. If it is not, you will get a compiler error during compile time.

Second, the variable cannot change data types. The following examples produce a compiler error during compile time.

#### C#

```
var v1 = 0 == 1;
```

```
v1 = "test";
```

#### Visual Basic

```
Dim v1 = 0 = 1
```

```
v1 = "test"
```

In the preceding example, you cannot change the variable `v1` from a `bool` to `string`, because the compiler first inferred the data type as a `bool`.

Third, the variable must be a local variable, not a parameter or return type to a method, or a field or property of a `class` or `struct`.

## Anonymous Types

You can use anonymous types to create a type that exposes only read-only properties without explicitly declaring the type. The type will be created as an object but will not have a name (thus, why it's called anonymous). Anonymous types also cannot have methods, fields, events, or writable properties. The following is an example of an anonymous type.

### C#

```
var person = new { LastName = "Alexander", FirstName = "Sean", MiddleInitial = "P" };

Console.WriteLine(person.LastName);
```

### Visual Basic

```
Dim person = New With {.LastName = "Alexander", .FirstName = "Sean", .MiddleInitial = "P"}

Console.WriteLine(person.LastName)
```

### Output

```
Alexander
```

Right now, you may be wondering how anonymous types can be useful if you cannot change their properties and cannot return them to the calling method. Later in this chapter, you will see how you can use LINQ to return a subset of data, or even a new data type created from parts of the data source.

## Object Initialization

You can initialize objects during their creation, as illustrated in the following examples.

### C#

```
Person person = new Person()
{
    LastName = "Alexander",
    FirstName = "Sean P.",
    Age = 46,
    Sex = Sex.Male
};
```

### Visual Basic

```
Dim person As New Person() With _
{
    .LastName = "Alexander",
    .FirstName = "Sean P.",
    .Age = 46,
    .Sex = Sex.Male
}
```

The more verbose, older way to initialize objects is as follows.

**C#**

```
Person person = new Person();
person.LastName = "Alexander";
person.FirstName = "Sean P.";
person.Age = 46;
person.Sex = Sex.Male;
```

**Visual Basic**

```
Dim person As New Person()
person.LastName = "Alexander"
person.FirstName = "Sean P."
person.Age = 46
person.Sex = Sex.Male
```

The usefulness of this shorter form of object initialization will become apparent in some of the following LINQ examples.

## Collection Initialization

Previous chapters used the following syntax to create a collection with default values.

**C#**

```
ArrayEx<int> array = new ArrayEx<int>(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 });
```

**Visual Basic**

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})
```

In Chapter 6, you went through the process of adding the interface *ICollection(T)* to your custom collection classes. With this interface, you can now create a collection instance with default items by using the following syntax instead.

**C#**

```
ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

**Visual Basic 2010**

```
Dim array As New ArrayEx(Of Integer) From {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

You can create and initialize any class that implements *ICollection(T)* in this same manner.

## Picking a Data Source (*from* Clause) and Selecting Results (*select* Clause)

The two basic operations of a LINQ query include picking your data source (the *from* clause) and selecting your results (the *select* clause).

## The *from* Clause

The *from* clause is the first part of the LINQ query and uses the following format.

### C# and Visual Basic

```
from (range variable) in (data source)
```

The data source must implement the *IEnumerable*, *IEnumerable(T)*, or *IQueryable(T)* interfaces. For collections, you should only be concerned with the *IEnumerable* and *IEnumerable(T)* interfaces. You can think of the *from* clause as a *foreach* statement written like the following.

### C#

```
foreach (range variable) in (datasource)
```

### Visual Basic

```
For each (range variable) in (datasource)
```

The difference between the *foreach* statement and the *from* clause is that the range variable in a *from* clause is used as a syntactic convenience and never actually stores the current item. However, you can use this variable in other clauses to interact with the retrieved information about the current item.

## The *select* Clause

The *select* clause is usually the last clause in a LINQ query. (The *group* clause appears after it, as discussed later.) This clause specifies what the results will contain when all clauses before it and any expressions in it are executed.

## Examples of *from* and *select* Clauses

The following example shows how the *select* clause returns the item in its expression.

### C#

```
ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var items = from element in array select element;
foreach (int element in items)
{
    Console.WriteLine(element);
}
```

### Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim items = From element In array Select element
For Each element As Integer In items
    Console.WriteLine(element)
Next
```

## Output

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

The code traverses the *array* variable and returns a collection that contains the elements in *array*. This is accomplished by specifying the range variable *element* in the *select* clause.

The following example demonstrates how to use the *select* clause to perform an operation on the range variable—in this case, a multiplication.

### C#

```
ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var doubled = from element in array select element * 2;
foreach (int element in doubled)
{
    Console.WriteLine(element);
}
```

### Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim doubled = From element In array Select element * 2
For Each element As Integer In doubled
    Console.WriteLine(element)
Next
```

## Output

```
2  
4  
6  
8  
10  
12  
14  
16  
18  
20
```

The preceding example returns a collection that contains each element in the array, multiplied by 2 because of the *element \* 2* operation included in the *select* clause.

You can also use the *select* clause to return a different type than the original data source contained, as shown in the following code.

**C#**

```
ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var items = from element in array select element.ToString();
foreach (string element in items)
{
    Console.WriteLine(element);
}
```

**Visual Basic**

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim items = From element In array Select (element.ToString())
For Each element As String In items
    Console.WriteLine(element)
Next
```

**Output**

```
1
2
3
4
5
6
7
8
9
10
```

This example returns a collection of *strings* instead of *ints*. This was accomplished by using *element.ToString()* in the *select* clause.

You can also use the *select* clause to return the fields or properties of an object instead of the object.

**C#**

```
Person []people = new Person[]
{
    new Person("Cristina", "Potra"),
    new Person("Jeff", "Phillips"),
    new Person("Mark", "Alexieff"),
    new Person("Holly", "Holt"),
    new Person("Gordon L.", "Hee"),
    new Person("Annie", "Herriman"),
};

var names = from person in people select person.LastName + ", " + person.FirstName;
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

## Visual Basic

```
Dim people As Person() = New Person() _
    { _
        New Person("Cristina", "Potra"), _
        New Person("Jeff", "Phillips"), _
        New Person("Mark", "Alexieff"), _
        New Person("Holly", "Holt"), _
        New Person("Gordon L.", "Hee"), _
        New Person("Annie", "Herriman") _
    }

Dim names = From person In people Select person.LastName + ", " + person.FirstName
For Each name As String In names
    Console.WriteLine(name)
Next
```

## Output

```
Potra, Cristina
Phillips, Jeff
Alexieff, Mark
Holt, Holly
Hee, Gordon L.
Herriman, Annie
```

The preceding code combines the person's last name with his or her first name and then returns a collection containing only that information. This was accomplished by specifying *person.LastName + ", " + person.FirstName* in the *select* clause.

You can create an anonymous type from another value type or object, as shown in the following code.

## C#

```
ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var doubled = from element in array select new
    { Value = element, IsEven = (element % 2 == 0), Doubled=element*2 };
foreach (var element in doubled)
{
    Console.WriteLine(element);
}
```

## Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim doubled = From element In array Select New With _
    {.Value = element, .IsEven = (element Mod 2 = 0), .Doubled = element * 2}
For Each element In doubled
    Console.WriteLine(element)
Next
```

**Output**

```
{ Value = 1, IsEven = False, Doubled = 2 }
{ Value = 2, IsEven = True, Doubled = 4 }
{ Value = 3, IsEven = False, Doubled = 6 }
{ Value = 4, IsEven = True, Doubled = 8 }
{ Value = 5, IsEven = False, Doubled = 10 }
{ Value = 6, IsEven = True, Doubled = 12 }
{ Value = 7, IsEven = False, Doubled = 14 }
{ Value = 8, IsEven = True, Doubled = 16 }
{ Value = 9, IsEven = False, Doubled = 18 }
{ Value = 10, IsEven = True, Doubled = 20 }
```

This example creates an anonymous type containing the properties *Value*, *IsEven*, and *Doubled* by specifying *new { Value = element, IsEven = (element % 2 == 0), Doubled=element\*2 }* in the *select* clause. Now you should be able to see how anonymous types can be useful.

## Filtering Results (the *where* Clause)

You can filter your results by using the *where* clause.

The following example filters the range element by checking to see if it is even.

**C#**

```
ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var even = from element in array where element % 2 == 0 select element;
foreach (int element in even)
{
    Console.WriteLine(element);
}
```

**Visual Basic**

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim even = From element In array Where element Mod 2 = 0 Select element
For Each element As Integer In even
    Console.WriteLine(element)
Next
```

**Output**

```
2
4
6
8
10
```

The preceding code returns only the even numbers by performing a modulo operation in the *where* clause.

You can filter a list of objects by using the object's fields, properties, or methods.

### C#

```
Person []people = new Person[]
{
    new Person("Cristina", "Potra", 22, Sex.Female),
    new Person("Jeff", "Phillips", 17, Sex.Male),
    new Person("Mark", "Alexieff", 55, Sex.Male),
    new Person("Holly", "Holt", 58, Sex.Female),
    new Person("Gordon L.", "Hee", 87, Sex.Male),
    new Person("Annie", "Herriman", 34, Sex.Female),
};

var names = from person in people
            where person.Age < 18
            select person.LastName + ", " + person.FirstName;
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

### Visual Basic

```
Dim people As Person() = New Person() _
{
    New Person("Cristina", "Potra", 22, Sex.Female), _
    New Person("Jeff", "Phillips", 17, Sex.Male), _
    New Person("Mark", "Alexieff", 55, Sex.Male), _
    New Person("Holly", "Holt", 58, Sex.Female), _
    New Person("Gordon L.", "Hee", 87, Sex.Male), _
    New Person("Annie", "Herriman", 34, Sex.Female) _
}

Dim names = From person In people _
            Where person.Age < 18 _
            Select person.LastName + ", " + person.FirstName
For Each name As String In names
    Console.WriteLine(name)
Next
```

### Output

Phillips, Jeff

The preceding code returns only minors by specifying *person.Age < 18* in the *where* clause.

## Ordering Results (the *orderby* Clause)

With the *orderby* clause, you can sort items in descending or ascending order using their default comparer.

The following example shows how you can sort a list of integers in descending order.

### C#

```
ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var numbers = from element in array orderby element descending select element;
foreach (int element in numbers)
{
    Console.WriteLine(element);
}
```

### Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim numbers = From element In array Order By element Descending Select element
For Each element As Integer In numbers
    Console.WriteLine(element)
Next
```

### Output

```
10
9
8
7
6
5
4
3
2
1
```

The preceding code sorts the list in descending order by specifying *element descending* in the *orderby* clause.

The following example shows how you can sort on multiple fields.

### C#

```
Person []people = new Person[]
{
    new Person("Cristina", "Potra", 22, Sex.Female),
    new Person("Jeff", "Phillips", 17, Sex.Male),
    new Person("Mark", "Alexieff", 55, Sex.Male),
    new Person("Holly", "Holt", 58, Sex.Female),
    new Person("Gordon L.", "Hee", 87, Sex.Male),
    new Person("Annie", "Herriman", 34, Sex.Female),
};
```

```
var names = from person in people
            orderby person.LastName, person.FirstName
            select person.LastName + ", " + person.FirstName;
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

## Visual Basic

```
Dim people As Person() = New Person() _
{ _
    New Person("Cristina", "Potra", 22, Sex.Female), _
    New Person("Jeff", "Phillips", 17, Sex.Male), _
    New Person("Mark", "Alexieff", 55, Sex.Male), _
    New Person("Holly", "Holt", 58, Sex.Female), _
    New Person("Gordon L.", "Hee", 87, Sex.Male), _
    New Person("Annie", "Herriman", 34, Sex.Female) _
}

Dim names = From person In people _
    Order By person.LastName, person.FirstName _
    Select person.LastName + ", " + person.FirstName
For Each name As String In names
    Console.WriteLine(name)
Next
```

## Output

```
Alexieff, Mark
Hee, Gordon L.
Herriman, Annie
Holt, Holly
Phillips, Jeff
Potra, Cristina
```

The preceding code sorts the returned values by specifying *person.LastName*, *person.FirstName* in the *orderby* clause. The *orderby* clause sorts by the *person.LastName* field, and then by *person.FirstName*. Also note how the comparison is performed on the specified fields, and not on the object itself. You cannot specify *person* because there is no default comparer for the *Person* struct.

## The *group* Clause

You can use the *group* clause to group the results based on a key. The *group* clause can be the last clause in the LINQ query.

The following example shows how you can group on the range variable.

### C#

```
ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var numbers = from element in array group element by element % 2 == 0;
foreach (IGrouping<bool, int> group in numbers)
```

```
{
    Console.WriteLine("{0}", group.Key ? "Even" : "Odd");
    foreach (int element in group)
    {
        Console.WriteLine("\t" + element);
    }
}
```

**Visual Basic**

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim numbers = From element In array Group element By Key = ((element Mod 2) = 0) Into Group
For Each group In numbers
    Console.WriteLine("{0}", If(group.Key, "Even", "Odd"))
    For Each element As Integer In group.Group
        Console.WriteLine(vbTab & element)
    Next
Next
```

**Output**

```
Odd
1
3
5
7
9
Even
2
4
6
8
10
```

The preceding code groups the numbers in an even and odd group by specifying *element%2==0* for the *group* clause.

You can also group on individual fields, as shown in the following example.

**C#**

```
Person []people = new Person[]
{
    new Person("Cristina", "Potra", 22, Sex.Female),
    new Person("Jeff", "Phillips", 17, Sex.Male),
    new Person("Mark", "Alexieff", 55, Sex.Male),
    new Person("Holly", "Holt", 58, Sex.Female),
    new Person("Gordon L.", "Hee", 87, Sex.Male),
    new Person("Annie", "Herriman", 34, Sex.Female),
};

var names = from person in people group person by person.Sex;
foreach (IGrouping<Sex, Person> group in names)
```

```
{  
    Console.WriteLine("{0}", group.Key);  
    foreach (Person person in group)  
    {  
        Console.WriteLine("\t{0}, {1}", person.LastName, person.FirstName);  
    }  
}
```

### Visual Basic

```
Dim people As Person() = New Person() _  
{ _  
    New Person("Cristina", "Potra", 22, Sex.Female), _  
    New Person("Jeff", "Phillips", 17, Sex.Male), _  
    New Person("Mark", "Alexieff", 55, Sex.Male), _  
    New Person("Holly", "Holt", 58, Sex.Female), _  
    New Person("Gordon L.", "Hee", 87, Sex.Male), _  
    New Person("Annie", "Herriman", 34, Sex.Female) _  
}  
  
Dim names = From person In people Group person By person.Sex Into Group  
For Each group In names  
    Console.WriteLine("{0}", group.Key)  
    For Each person As Person In group.Group  
        Console.WriteLine(vbTab & "{0}, {1}", person.LastName, person.FirstName)  
    Next  
Next
```

### Output

```
Female  
    Potra, Cristina  
    Holt, Holly  
    Herriman, Annie  
Male  
    Phillips, Jeff  
    Alexieff, Mark  
    Hee, Gordon L.
```

The preceding code groups the people according to their sex by specifying *person.Sex* for the *group* clause.

## The *join* Clause

With the *join* clause, you can join two different collections into one collection. To accomplish the *join*, both collections must contain a value that can be compared for equality.

### Inner Join

Inner joins produce a result by combining multiple collections into one. The result of the query is the expression in the *select* clause. Each item in the data source is traversed and combined with the item in the join collection that matches the join criteria. Items that do not match are not returned.

Here's how to perform an inner join.

### C#

```
ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var integerTypes = from value in Enum.GetValues(typeof(IntegerType)).Cast<int>()
                   select new { Id = value, Name = ((IntegerType)value).ToString() };

var joined = from element in array join e in integerTypes on element % 2 equals e.Id
               select new { Value = element, Type=e.Name };
foreach (var tmp in joined)
{
    Console.WriteLine(tmp);
}
```

### Visual Basic

```
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim integerTypes = From value In [Enum].GetValues(GetType(IntegerType)).Cast(Of Integer)() _
                  Select New With {.Id = value, .Name = DirectCast(value, IntegerType).
ToString()}

Dim joined = From element In array Join e In integerTypes On element Mod 2 Equals e.Id _
            Select New With {.Value = element, .Type = e.Name}
For Each tmp In joined
    Console.WriteLine(tmp)
Next
```

### Output

```
{ Value = 1, Type = Odd }
{ Value = 2, Type = Even }
{ Value = 3, Type = Odd }
{ Value = 4, Type = Even }
{ Value = 5, Type = Odd }
{ Value = 6, Type = Even }
{ Value = 7, Type = Odd }
{ Value = 8, Type = Even }
{ Value = 9, Type = Odd }
{ Value = 10, Type = Even }
```

For demonstration purposes, *integerTypes* stores a collection of anonymous types that represent the following *IntegerType* enumeration.

### C#

```
public enum IntegerType
{
    Even = 0,
    Odd = 1
}
```

## Visual Basic

```
Public Enum IntegerType
    Even = 0
    Odd = 1
End Enum
```

This is accomplished by storing the integer form of *IntegerType* as an *Id* and the string form of *IntegerType* as the *Name*. The two collections are inner joined by comparing the *element%2* to the *Id* field in *integerTypes*. The enumeration was written so that *element%2* contains the same value as *IntegerType.Even* when the number is even and *IntegerType.Odd* when the number is odd. You could instead create a data type and collection for holding the information in *integerTypes*.

## Outer Join

An outer join operates the same way as an inner join except it also returns results for items in the data source that do not have a matching item in the join collection.

The following example shows how to do a left outer join.

### C#

```
var primes = from value in new ArrayEx<int>() { 2, 3, 5, 7 }
            select new { Id = value, Name = "Prime" };

ArrayEx<int> array = new ArrayEx<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var joined = from element in array
            join prime in primes on element equals prime.Id
            into primeGroup
            from subprime in primeGroup.DefaultIfEmpty()
            select new
            {
                Value=element,
                Type = (subprime == null ? "Not Prime" : subprime.Name)
            };

foreach (var tmp in joined)
{
    Console.WriteLine(tmp);
}
```

### Visual Basic

```

Dim primes = From value In New ArrayEx(Of Integer)(New Integer() {2, 3, 5, 7}) _
    Select New With {.Id = value, .Name = "Prime"}
Dim array As New ArrayEx(Of Integer)(New Integer() {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

Dim joined = From element In array _
    Group Join prime In primes On element Equals prime.Id _
    Into primeGroup = Group _
    From subprime In primeGroup.DefaultIfEmpty() _
    Select New With _
    {
        .Value = element,
        .Type = If(subprime Is Nothing, "Not Prime", subprime.Name)
    }

For Each tmp In joined
    Console.WriteLine(tmp)
Next

```

### Output

```

{ Value = 1, Type = Not Prime }
{ Value = 2, Type = Prime }
{ Value = 3, Type = Prime }
{ Value = 4, Type = Not Prime }
{ Value = 5, Type = Prime }
{ Value = 6, Type = Not Prime }
{ Value = 7, Type = Prime }
{ Value = 8, Type = Not Prime }
{ Value = 9, Type = Not Prime }
{ Value = 10, Type = Not Prime }

```

The preceding code executes a prime query to demonstrate the left join. An inner join is then executed at the line *from element in array join prime in primes on element equals prime.Id*. The *into primeGroup* stores the results of the call into a temporary identifier named *primeGroup*. The *primeGroup.DefaultIfEmpty ()* is then used as the data source. The *DefaultIfEmpty* method will return a default value, *null* in this case, if a matching item in the join table cannot be located. The *select* clause checks to see if the item is *null* and if so, returns “*Not Prime*”; otherwise, it returns *subprime.Name*.

## The *let* Clause

With the *let* clause, you can store subexpressions that you can later use in subsequent clauses.

### C#

```

Person[] people = new Person[]
{
    new Person("Cristina", "Potra", 22, Sex.Female),
    new Person("Jeff", "Phillips", 17, Sex.Male),
    new Person("Mark", "Alexieff", 55, Sex.Male),
}

```

```

        new Person("Holly", "Holt", 58, Sex.Female),
        new Person("Gordon L.", "Hee", 87, Sex.Male),
        new Person("Annie", "Herriman", 34, Sex.Female),
    };

var peopleWithIName = (from person in people
    let chars = (person.FirstName + person.LastName).ToCharArray()
    from ch in chars where ch == 'i' || ch == 'I' select
    person).Distinct();

foreach (Person person in peopleWithIName)
{
    Console.WriteLine("The name {0}, {1} contains an 'i'"
                      , person.LastName, person.FirstName);
}

```

### Visual Basic

```

Dim people As Person() = New Person() _
{ _
    New Person("Cristina", "Potra", 22, Sex.Female), _
    New Person("Jeff", "Phillips", 17, Sex.Male), _
    New Person("Mark", "Alexieff", 55, Sex.Male), _
    New Person("Holly", "Holt", 58, Sex.Female), _
    New Person("Gordon L.", "Hee", 87, Sex.Male), _
    New Person("Annie", "Herriman", 34, Sex.Female) _
}

Dim peopleWithIName = (From person In people _
    Let chars = (person.FirstName + person.LastName).ToCharArray() _
    From ch In chars _
    Where ch = "i"c Or ch = "I"c _
    Select person).Distinct()

For Each person As Person In peopleWithIName
    Console.WriteLine("The name {0}, {1} contains an 'i'" _
                      , person.LastName _
                      , person.FirstName)

```

Next

### Output

```

The name Potra, Cristina contains an 'i'
The name Phillips, Jeff contains an 'i'
The name Alexieff, Mark contains an 'i'
The name Herriman, Annie contains an 'i'

```

The preceding code checks each first name for the letter *i* and then selects the range variable if it contains *i*. You could also accomplish this without using the *let* clause and simply doing the following.

### C#

```

var peopleWithIName = (from person in people
    where (person.FirstName +
    person.LastName).ToLower().Contains("i")
    select person);

```

### Visual Basic

```
Dim peopleWithIName = (From person In people _
    Where (person.FirstName + _
    person.LastName).ToLower().Contains("i") _
    Select person)
```

There are several things to note about the example that uses the *let* clause. First, notice that the *Distinct* method is called on the results. The *let* clause splits the full name into an array of *chars*. The array is then used as a data source, which causes the *select* clause to execute each time it finds an *i*. The *Distinct* method removes all duplicates from the results. Therefore, when a name contains more than one *i*, the *Distinct* method will remove the duplicates caused by repeated execution of the *select* clause. You could write the LINQ statement as the following.

### C#

```
var peopleWithIName = from person in people
    let chars = (person.FirstName + person.LastName).ToCharArray()
    where
        ((from ch in chars where ch == 'i' ||
        ch == 'I' select ch).Count() > 0)
    select person;
```

### Visual Basic

```
Dim peopleWithIName = From person In people _
    Let chars = (person.FirstName + person.LastName).ToCharArray() _
    Where _
        ((From ch In chars Where ch = "i"c Or -
        ch = "I"c Select ch).Count() > 0) _
    Select person
```

The *where* clause is now executed using a nested query on the subexpression created by the *let* clause. The *Count* method is used to count the occurrences of *i*.

## Summary

In this chapter, you learned about LINQ and how to use it. As described, LINQ gives you an exciting new way to write queries against collections and other data sources. LINQ statements take less time to write than natural language statements and are very powerful.

## Chapter 8

# Using Threads with Collections

After completing this chapter, you will be able to

- Understand what a thread is.
- Understand why you should synchronize access to collections.
- Identify what tools are available to help you synchronize access to collections and when to use each one.
- Add synchronization support to your custom collection classes.

## What Is a Thread?

Windows is a multitasking operating system. From the user's point of view, this means that multiple processes seem to run at the same time. From the system viewpoint, multitasking is far more complex than that. You can find many books that go into the inner workings of processes and multitasking, but doing so is beyond the scope of this book. Instead, this section provides only the most basic explanation.

A computer running Windows runs far more processes than it has processors, so it is impossible for the operating system to run all the processes at the same time. Instead, the operating system switches processes quickly among the available processors in a way that makes it *appear* to run all of the processes at the same time. In general, the faster the operating system can switch processes, the more responsive the overall system is. To switch the processes in and out, the operating system must maintain data (context) on each process, such as where it was when it was switched out code that was executing at that point. To understand threads, think of your process as being an operating system and your threads as processes running inside of your operating system. Each process can consist of multiple threads that can execute code within that process. (Each thread is still controlled by the real operating system, but each process is allowed to suggest to the real operating system when the thread should run.)

You may have realized that each thread requires memory for managing the thread and process cycles to switch the thread in and out. This means that the more threads you add, the more resources you use up. So, for performance reasons, you should create threads only when needed. Thread pools and synchronization objects, which can help you with performance and deadlock, are discussed later in this chapter.

## What Is Thread Synchronization?

Threads communicate with each other through *shared objects*. Shared objects can be read and written to by any of the threads. *Thread synchronization* is the process of synchronizing access to shared objects so that one thread doesn't read from the shared object while another thread is writing to it.

## Why Should I Care About Thread Synchronization?

The operating system controls when a thread wakes up; however, you can influence that somewhat through the use of thread sleep commands, events, and so on. Some methods are more along the lines of demands than recommendations, but ultimately, all threads are controlled by the operating system and the Microsoft .NET Framework—not your code. Look at the following example that uses the *ReverseArray* and *SumArray* methods.

C#

```
static void ReverseArray(object state)
{
    for ( ; ; )
    {
        ArrayEx<int> array = state as ArrayEx<int>;
        for (int i = 0; i < array.Count / 2; ++i)
        {
            int tmp = array[i];
            array[i] = array[array.Count - 1 - i];
            array[array.Count - 1 - i] = tmp;
        }
        Thread.Sleep(100);
    }
}

static void SumArray(object state)
{
    for ( ; ; )
    {
        ArrayEx<int> array = state as ArrayEx<int>;

        int sum = 0;
        Console.WriteLine("Summing array");
        for (int i = 0; i < array.Count; ++i)
        {
            sum += array[i];
        }
        if (s_sum != sum)
        {
            Console.WriteLine("Calculated sum was {0} instead of {1}", sum, s_sum);
        }
        else
```

```
{  
    Console.WriteLine("Sum is {0}", sum);  
}  
Thread.Sleep(100);  
}  
}
```

### Visual Basic

```
Sub ReverseArray(ByVal state As Object)  
    While (True)  
        Dim array As ArrayEx(Of Integer) = CType(state, ArrayEx(Of Integer))  
  
        For i As Integer = 0 To (array.Count / 2) - 1  
            Dim tmp As Integer = array(i)  
            array(i) = array(array.Count - 1 - i)  
            array(array.Count - 1 - i) = tmp  
        Next  
        Thread.Sleep(100)  
    End While  
End Sub  
  
Sub SumArray(ByVal state As Object)  
    While (True)  
        Dim array As ArrayEx(Of Integer) = CType(state, ArrayEx(Of Integer))  
  
        Dim sum As Integer = 0  
        Console.WriteLine("Summing array")  
        For i As Integer = 0 To array.Count - 1  
            sum += array(i)  
        Next  
        If (s_sum <> sum) Then  
            Console.WriteLine("Calculated sum was {0} instead of {1}", sum, s_sum)  
        Else  
            Console.WriteLine("Sum is {0}", sum)  
        End If  
        Thread.Sleep(100)  
    End While  
End Sub
```

The *ReverseArray* method simply reverses the specified array and the *SumArray* method sums the values of the array. Each method sleeps for approximately 100 milliseconds (ms) before repeating the operation. The *SumArray* method also checks the calculated sum against the true sum to verify that it calculated the sum correctly.

The code in the rest of this section uses the *ReverseArray* method to simulate a writer thread and the *SumArray* method to simulate a reader thread.

The following code creates a thread to execute the *SumArray* and *ReverseArray* methods on the variable *array*.

**C#**

```
ArrayEx<int> array = new ArrayEx<int>();

s_sum = 0;
for (int i = 0; i < 10; ++i)
{
    array.Add(i);
    s_sum += i;
}

ThreadPool.QueueUserWorkItem(new WaitCallback(ReverseArray), array);
ThreadPool.QueueUserWorkItem(new WaitCallback(SumArray), array);

Thread.Sleep(3000);
```

**Visual Basic**

```
Dim array As ArrayEx(Of Integer) = New ArrayEx(Of Integer)

s_sum = 0
For i As Integer = 0 To 9
    array.Add(i)
    s_sum += i
Next

ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf ReverseArray), array)
ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf SumArray), array)

Thread.Sleep(3000)
```

**Output**

```
Summing array
Sum is 45
Summing array

...
Sum is 45
Summing array
```

The variable *array* is assigned ten numbers from 0 through 9 and a true sum is calculated and stored in the static field *s\_sum* so that the *SumArray* method can verify its results. On my machine, both threads are able to execute without any noticeable problems. On the other hand, changing the 10 to 10000 or the 9 to 9999 causes the following output on my machine.

### Output

```
Summing array
Sum is 49995000
Summing array
Calculated sum was 59115551 instead of 49995000
Summing array
Calculated sum was 38547424 instead of 49995000
Summing array
Calculated sum was 61658817 instead of 49995000
Summing array
Calculated sum was 37244712 instead of 49995000
Summing array
Calculated sum was 57821468 instead of 49995000
Summing array
Calculated sum was 40425072 instead of 49995000
Summing array
Calculated sum was 60709149 instead of 49995000
Summing array
Calculated sum was 39022363 instead of 49995000
Summing array
Calculated sum was 62030664 instead of 49995000
Summing array
Calculated sum was 36669912 instead of 49995000
Summing array
Calculated sum was 62146863 instead of 49995000
Summing array
Calculated sum was 39314880 instead of 49995000
Summing array
Calculated sum was 64503512 instead of 49995000
Summing array
Calculated sum was 46560888 instead of 49995000
Summing array
Calculated sum was 59602675 instead of 49995000
Summing array
```

...

Your machine results are likely to differ because they depend upon your processor(s), anything else running in the background of your machine, and so on. Please note that the point of this exercise is to show that you cannot run a simple case to see if you should synchronize data to a thread. Instead, you should just do it! I have seen many companies fight countless bugs in their production systems because their developers misunderstood how and when threads read and write data. Many developers will tell you that threading problems are the most difficult bugs to track down.

## Why Not Write Thread Synchronization Code As Needed?

Thread synchronization requires careful thought. A thread that doesn't synchronize correctly can cause threads to read data before a write is complete or to deadlock (see the note at the end of this section). Take a look at the following example.

### Pseudocode

```
While waiting for Collection Update Event
    Lock
        Process collection
    Unlock
    Reset Collection Update Event
Loop
```

The preceding code resets the collection update event after the collection is unlocked. It is possible that a thread could update the collection and fire the event between the unlock and reset event. If this happens, the current thread will have to wait for another event to happen because the thread accidentally reset the current event.

Also look at the next example.

### Pseudocode

```
Thread A
Lock C
    Process some data (Takes 5 seconds)
Lock D
    Process Some Data
Unlock D
Unlock C

Thread B
Lock D
    Process some data (Takes 5 seconds)
Lock C
    Process Some Data
Unlock C
Unlock D
```

The preceding example causes Thread A to deadlock on Lock D because Thread B has locked Lock D and will not unlock it until it obtains Lock C—which is locked by Thread A.

With some locking mechanisms, you have to be aware of recursive locking. Take a look at the following example.

### Pseudocode

```
Class Collection
    Method Add(item)
        Lock C
            Process Collection
        Unlock C

Thread A
    Lock C
        collection.Add(item)
    Unlock C
```

The preceding example locks Lock C and then adds an item to the collection, which in return locks Lock C again. In such cases, some locking mechanisms will throw an exception on a recursive lock, whereas others may ignore the double lock, because both lock the same thread. You will need to read the documentation on MSDN to learn about recursive locking.



**Note** A deadlock occurs when a thread is waiting on a continue condition that never happens, possibly because the thread that is supposed to set the continue condition has also deadlocked, has neglected to set the continue condition, or has crashed. The continue condition might be anything from a lock (*Mutex*, *Semaphore*, and so on) to a variable (such as *boolean*, or *int*).

## .NET Framework Tools for Synchronization

In the .NET Framework, you can use three categories of objects to control data in threads: interlocked operations, signaling, and locking.

### Interlocked Operations

You can perform atomic operations on variables of types such as *Int32* and *Int64* by using the static methods in the *Interlocked* class. The *Interlocked* class helps guarantee that variable modifications are performed as one transactional operation. You would not normally use interlocked operations for collections, because when updating a collection, you typically update more than just the count, and you also have to ensure that no other operations can be performed on the collection while you are updating its internal data storage. Therefore, this book does not discuss the *Interlocked* class in detail, but it's mentioned here so you'll be aware of it for other cases where it may come in handy, such as reference counting or creating custom locks. You will see a brief description of the *CompareExchange* method later in this chapter in the "Getting Started" section in "Adding Synchronization Support to Your Collections Classes," which shows how to use the *CompareExchange* method to add synchronization support to custom collection classes.

## Signaling

*Signaling* is a way of allowing a thread to wait for a condition. The condition is defined using the *WaitHandle* class. You can use signaling to improve the performance of your application.

Suppose you have a thread that processes data only when items are added to a collection. An inefficient way of doing this would be as follows. Please note that locking is left out of the following example.

**C#**

```
while (bRunning )  
{  
  
    if (collections.Count > 0)  
    {  
        // Process all data  
  
        collections.Clear();  
    }  
  
    Thread.Sleep(100);  
}
```

**Visual Basic**

```
While (bRunning)  
  
    If (collections.Count > 0) Then  
        ' Process all data  
  
        collections.Clear()  
    End If  
  
    Thread.Sleep(100)  
End While
```

The code spins in an infinite loop, waiting for an item to be added to the collection. The *Thread.Sleep* method is called in the code to help with performance. Although the code doesn't contain a lot of statements, the operating system can swap the thread in and out at any moment. In contrast, look at the next example (which also leaves out locking).

**C#**

```
while (bRunning )  
{  
  
    itemsAvailable.WaitOne();  
  
    // Process all data  
  
    collections.Clear();  
  
    itemsAvailable.Reset();  
}
```

**Visual Basic**

```
While (bRunning)

    itemsAvailable.WaitOne()

    ' Process all data

    collections.Clear()

    itemsAvailable.Reset()
End While
```

The *itemsAvailable* variable is defined as the following.

**C#**

```
ManualResetEvent itemsAvailable = new ManualResetEvent(false);
```

**Visual Basic**

```
Dim itemsAvailable As ManualResetEvent = New ManualResetEvent(False)
```

The code no longer needs to call *Thread.Sleep* because it executes only when other code calls *itemsAvailable.Set()*. At that point, it clears the collection, and then calls the *Reset* method on the *ManualResetEvent* class so that the next call to *WaitOne* will not result in a false *true*. The downside to this example is that it doesn't provide a way to tell the thread to wake up for other cases, such as when the thread should shut down gracefully, or if you wanted to have the thread deal with multiple lists. Luckily the *WaitHandle* class has the *WaitAll* and *WaitAny* methods. The *WaitAll* method waits for *all* the specified elements to be signaled, whereas the *WaitAny* method waits for *any* of the elements to be signaled before resuming execution. The *WaitAny* method is perfect if you also want to be able to wake the thread up for a shutdown.

**C#**

```
for ( ; ; )
{
    int wakeup = WaitHandle.WaitAny(conditions);

    if (wakeup == 0)
    {
        break;
    }

    if (wakeup == 1)
    {
        // Process all data
        collections.Clear();
    }
}
```

**Visual Basic**

```
Dim wakeup As Integer = WaitHandle.WaitAny(conditions)
While (True)
    If (wakeup = 0) Then
        Exit While
    End If

    If (wakeup = 1) Then
        ' Process all data
        collections.Clear()
    End If
End While
```

The wait conditions are defined in the *conditions* variable as the following.

**C#**

```
WaitHandle []conditions = new WaitHandle[]
{
    new ManualResetEvent(false), // Shutdown
    new AutoResetEvent(false), // Items Available
};
```

**Visual Basic**

```
Dim conditions() As WaitHandle = New WaitHandle() _
{
    New ManualResetEvent(False),
    New AutoResetEvent(False)
}
```

The *WaitAny* method returns the index of the item that was signaled. You could also use an *enum* or a *const* instead of integer values to make it easier to read. In this case, the *itemsAvailable*.*Reset* event is changed to an *AutoResetEvent*. The *AutoResetEvent* automatically resets the event.

## Locking

Locks give control to a specified number of threads at a time. When a lock allows access to only one thread at a time, it is an *exclusive lock*.

### Exclusive Locks

With the *Mutex* and *Monitor* classes, you can define a lock with exclusive access. The *Mutex* class is a heavier lock that also allows locking across processes, whereas a *Monitor* class can only be used inside of the *AppDomain*. The *Mutex* class is also derived from *WaitHandle*, which allows it to be used with the *WaitHandle* methods. Microsoft Visual Basic and C# allow easy access to the *Monitor* class with the *SyncLock* and *lock* statements respectively. The following code examples use the *Monitor* class instead of the *Mutex* class because:

- Process synchronization is not needed.
- *WaitHandle* support is not needed.
- The *Monitor* class has built-in syntax.
- The additional overhead of a Mutex is not needed.

Monitors can be used with native syntax as follows.

#### C#

```
lock (locking object)
{
    // Critical code section
}
```

#### Visual Basic

```
SyncLock [locking object]
    ' Critical code section
End SyncLock
```

The preceding statement is the same as writing the following.

#### C#

```
System.Threading.Monitor.Enter(locking object);
try
{
    // Critical code section
}
finally
{
    System.Threading.Monitor.Exit(locking object);
}
```

#### Visual Basic

```
System.Threading.Monitor.Enter(locking object)
Try
    ' Critical code section
Finally
    System.Threading.Monitor.Exit(locking object)
End Try
```

The *Enter*, *Wait*, and *TryEnter* methods are used to acquire the lock, whereas the *Exit* and *Wait* methods are used to release the lock. All methods take as an argument the object to acquire or release the lock on.



**Note** You should not use *Me* in Visual Basic or *this* in C# as an argument to the *Monitor* methods. Instead, you should create an object and use that instead. Everyone has access to *Me* and *this* and can use them as a lock however they like. For example, you can guarantee that the object you create as a lock for reading can only be used for reading. Doing it this way can eliminate some potential deadlock problems later on.

You will learn how to use the *Monitor* class with collections in the section “Using the *Monitor Class*,” later in this chapter.

## Non-Exclusive Locks

The *Semaphore*, *ReaderWriterLock*, and *ReaderWriterLockSlim* classes allow access to multiple threads.

You can use the *Semaphore* class to define how many threads can have access before other calling threads are blocked. After a thread releases the semaphore, another thread can take its place.

Collection classes allow two types of access: reading and writing. An infinite number of threads can read the collection simultaneously, but only one thread can write to it safely at one time—and only then if no threads are reading the collection at that point. You must also consider that you don’t want to “starve” your writing threads. Starvation can occur because multiple threads can read but only one thread can write at a time. Rather than always granting a thread read access when another thread is currently reading, you must balance the system if another thread is waiting to write. Luckily, the .NET Framework defined a *ReaderWriterLock* and *ReaderWriterLockSlim* class for you to use instead of creating your own.

Later in this chapter, in the section “Using the *ReaderWriterLockSlim* Class,” you will learn how to use the *ReaderWriterLockSlim* class with collections.



**Note** *ReaderWriterLockSlim* has all of the functionality of *ReaderWriterLock* but provides simplified rules for recursions and for upgrading and downgrading locks. These new changes should increase the performance of the locks, and also avoid some deadlocking cases. For this reason, MSDN recommends that you use *ReaderWriterLockSlim* for all new development. So, this book uses only *ReaderWriterLockSlim*.

## Adding Synchronization Support to Your Collection Classes

It would be useful for the user of the collection classes you created in the first three chapters to have access to your collections in a multithreaded application. The following sections show how to add collection support to the custom classes you created in previous chapters.

## *ICollection* Revisited

In Chapter 6, ".NET Collection Interfaces," you learned about the *ICollection* interface and how to add it to your custom collection classes. The implementation of the *IsSynchronized* and *SyncRoot* properties were left to this chapter. These two properties are used for synchronization.

The *IsSynchronized* property states whether the properties and methods for the collection are thread safe. In the "Implementing a Synchronized Wrapper Class" section later in this chapter, you discover how to create a class that wraps your custom class to make it synchronized. In that case, you need to return a *true* to state that the class is synchronized and guarantee that each method and property that interacts with the collection is thread safe. In the "SyncRoot vs. the Synchronized Wrapper Class (*IsSynchronized*)" section later in this chapter, you see the advantages and disadvantages of both.

The *SyncRoot* property returns an object that can be used to synchronize access to the collection. The return object is used with the *lock* statement and *Monitor* class.

## Getting Started

In this section, you will modify the classes you modified in Chapter 6 to support synchronization, or you can look at the examples in the folder for Chapter 7, "Introduction to LINQ." For each collection type, you need to modify the file with the name <TypeName>.Collection.cs for C# or <TypeName>.Collection.vb for Visual Basic. Fortunately, the *SyncRoot* and *IsSynchronized* properties in all the classes are implemented in the same way. In the file you will find the *IsSynchronized* and *SyncRoot* properties. You will not be adding synchronization support directly to the methods and properties, so you will not need to modify the *IsSynchronized* property. Instead, you will allow users to control the synchronization through the *SyncRoot* property. To do this, each class needs to change the following code.

### C#

```
object System.Collections.ICollection.SyncRoot
{
    get { throw new NotImplementedException(); }
}
```

### Visual Basic

```
Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property
```

Here's what the code should be changed to.

#### C#

```
object m_syncRoot;
object System.Collections.ICollection.SyncRoot
{
    get
    {
        if (m_syncRoot == null)
        {
            System.Threading.Interlocked.CompareExchange(ref m_syncRoot, new object(), null);
        }
        return m_syncRoot;
    }
}
```

#### Visual Basic

```
Private m_syncRoot As Object
Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
    Get
        If (m_syncRoot Is Nothing) Then
            System.Threading.Interlocked.CompareExchange(m_syncRoot, New Object(), Nothing)
        End If
        Return m_syncRoot
    End Get
End Property
```

The preceding code uses the *Interlocked.CompareExchange* method to create the *SyncRoot* object. The *CompareExchange* method sets *m\_syncRoot* to *new object()* if *m\_syncRoot* is *null*. Doing this synchronizes the reading and writing of *m\_syncRoot* across threads. Remember, you cannot use *m\_syncRoot* because it may be set to *null*. You could instead create *m\_syncRoot* in the constructor; however, that carries the disadvantage that every collection would contain a *SyncRoot* object even if it were never used.

Now you can synchronize your collection classes using the following code.

#### C#

```
ArrayEx<int> list = new ArrayEx<int>();

lock (((ICollection)list).SyncRoot)
{
    list.Add(12);
}
```

#### Visual Basic

```
Dim list As ArrayEx(Of Integer) = New ArrayEx(Of Integer)

SyncLock CType(list, ICollection).SyncRoot
    list.Add(12)
End SyncLock
```

Here's an alternative approach that you can use.

#### C#

```
ArrayEx<int> list = new ArrayEx<int>();

if (System.Threading.Monitor.TryEnter(((ICollection)list).SyncRoot))
{
    try
    {
        list.Add(12);
    }
    finally
    {
        System.Threading.Monitor.Exit(((ICollection)list).SyncRoot);
    }
}
```

#### Visual Basic

```
If (System.Threading.Monitor.TryEnter(CType(list, ICollection).SyncRoot)) Then
    Try
        list.Add(12)
    Finally
        System.Threading.Monitor.Exit(CType(list, ICollection).SyncRoot)
    End Try
End If
```

## SyncRoot vs. the Synchronized Wrapper Class (*IsSynchronized*)

One advantage of using the Synchronized Wrapper is that it forces users to use synchronization, as illustrated in the next example.



**Note** The information in this section for a Synchronized Wrapper class also holds true for other collection classes that return *true* for the *IsSynchronized* property. Refer to the "Implementing a Synchronized Wrapper Class" section later in this chapter for more information about Synchronized Wrapper classes.

#### C#

```
test.Add(item);
```

#### Visual Basic

```
test.Add(item)
```

Here's an example that uses *SyncRoot* instead.

**C#**

```
lock (((ICollection)test).SyncRoot)
{
    test.Add(item);
}
```

**Visual Basic**

```
SyncLock CType(test, ICollection).SyncRoot
    test.Add(item)
End SyncLock
```

With *SyncRoot*, all developers must wrap their calls with a *lock* statement. The Synchronized Wrapper handles this for the user. One drawback to using the Synchronized Wrapper is that synchronization occurs within the synchronized methods and properties only, and not in between. Take a look at the following example of a Synchronized Wrapper.

**C#**

```
object item = null;
if (test.Count > 0)
{
    // The collection may no longer contain values
    item = test[0];
}
// Process item
```

**Visual Basic**

```
Dim item As Object = Nothing
If (test.Count > 0) Then
    ' The collection may no longer contain values
    item = test(0)
End If
' Process item
```

Now look at the following example that uses *SyncRoot*.

**C#**

```
object item = null;
lock (((ICollection)test).SyncRoot)
{
    if (test.Count > 0)
    {
        item = test[0];
    }
}
// Process item
```

**Visual Basic**

```
Dim item As Object = Nothing
If (test.Count > 0) Then
    ' The collection may no longer contain values
    item = test(0)
End If
' Process item
```

Using synchronized wrappers does not guarantee that *Count* will not change before you get the first item. One way to handle this problem would be to create another method that combines the operations. This is possible only if you created the class, because adding an extensible function doesn't safeguard you against the original developer changing the way she creates locks. This may also substantially increase the number of methods, and may cause developers to add multiple methods with different names that do the same thing.

The chances of 3 increase when you use *SyncRoot*, as shown in the following examples. The first example uses a Synchronized Wrapper.

#### C#

```
b.Add(a[0]);
a.RemoveAt(0);
```

#### Visual Basic

```
b.Add(a(0))
a.RemoveAt(0)
```

Here's the example using *SyncRoot*.

#### C#

```
List<object> a = new List<object>();
List<object> b = new List<object>();
lock (((ICollection)a).SyncRoot)
{
    lock (((ICollection)b).SyncRoot)
    {
        b.Add(a[0]);
    }

    a.RemoveAt(0);
}
```

#### Visual Basic

```
Dim a As List(Of Integer) = New List(Of Integer)
Dim b As List(Of Integer) = New List(Of Integer)

SyncLock CType(a, ICollection).SyncRoot
    SyncLock CType(b, ICollection).SyncRoot
        b.Add(a(0))
    End SyncLock
    a.RemoveAt(0)
End SyncLock
```

The user locked the *b* collection while in *a* collection lock. Any thread that locks the *a* collection while in the *b* collection lock while this code is running will deadlock. It is easy to see this in the preceding code but not when the nested lock happens later in a function call stack. You could fix the nested problem by doing a *TryEnter* instead, but now you get into what to do when the nested lock fails. Again, in the preceding example, you can easily figure it out,

but not in the production code where the nested lock happens after numerous events, function calls, and property changes have occurred—at which point it might be time to finally convince management that designing by compiling isn't the way to go.

Because *SyncRoot* provides more control to users, users are able to handle performance problems.

Here's an example using a Synchronized Wrapper.

**C#**

```
a.AddRange(items);
```

**Visual Basic**

```
a.AddRange(items)
```

And here's an example using *SyncRoot*.

**C#**

```
List<object> a = new List<object>();  
  
lock (((ICollection)a).SyncRoot)  
{  
    foreach (object item in items)  
    {  
        a.Add(item);  
    }  
}
```

**Visual Basic**

```
Dim a As List(Of Integer) = New List(Of Integer)  
SyncLock CType(a, ICollection).SyncRoot  
    For Each item In items  
        a.Add(item)  
    Next  
End SyncLock
```

Using the *SyncRoot* property also means that users have to know what they are doing. Holding a lock should be as short as possible. Adding an *AddRange* method to your Synchronized Wrapper class would enable you to add a number of items, release the lock, acquire the lock again, add some more items, release the lock, and so on. This would be handy if the user specified a very large number of items to add. With some education, users could do it with the *SyncRoot* call instead.

In general, Synchronized Wrappers are good when you want to have complete control over how synchronization is done, want limited access to the wrapped collection, or want synchronization to be transparent to the user of the collection. But with a little education, you could teach users how to use the *SyncRoot* property and how to be aware of threading issues.

Chances are that you will use the *SyncRoot* property the majority of the time, but think about

the following scenario: You have an internal collection of objects called *InternalObjects* that is stored in the *MySingleton* object and is dynamically updated. For some reason, you decide to provide the users with a collection of publicly accessible objects, called *PublicObject*. Users can access the collection of *PublicObjects*, which could be updated because of changes to the collection of *InternalObjects*. You may find a need to lock both collections if either collection is locked, or you may need a more complex locking mechanism that gives higher priority to the collection of *InternalObjects*. Implementing synchronization in the *InternalObjects* and *PublicObject* collection classes would eliminate the need for users to learn your complex synchronization procedures.

## Using the *Monitor* Class

The easiest way to use the *Monitor* class is through the built-in language syntax, such as the following.

### C#

```
lock (locking object)
{
    // Critical code section
}
```

### Visual Basic

```
SyncLock [locking object]
    ' Critical code section
End SyncLock
```

As discussed before, the preceding statement is the same as writing the following.

### C#

```
System.Threading.Monitor.Enter(locking object);
try
{
    // Critical code section
}
finally
{
    System.Threading.Monitor.Exit(locking object);
}
```

### Visual Basic

```
System.Threading.Monitor.Enter(locking object)
Try
    ' Critical code section
Finally
    System.Threading.Monitor.Exit(locking object)
End Try
```

The advantage of using the built-in language syntax is that developers can easily read and understand what you did. The code also handles the possibility of an exception being thrown. However, the code doesn't allow you to time out on obtaining a lock because it uses the *Enter* method instead of the *TryEnter* method. The *TryEnter* method takes the object you want to acquire a lock on as an argument. If the lock is obtained, it returns *true*, and if not, it returns *false*. A *false* return means that another thread has acquired the lock. Here's an example.

**C#**

```
if (System.Threading.Monitor.TryEnter(locking object))
{
    try
    {
        // Critical Code Section
    }
    finally
    {
        System.Threading.Monitor.Exit(locking object);
    }
}
else
{
    // Failed to obtain lock
}
```

**Visual Basic**

```
If (System.Threading.Monitor.TryEnter(locking object)) Then
    Try
        ' Critical code section
    Finally
        System.Threading.Monitor.Exit(conditions)
    End Try
Else
    ' Failed to obtain lock
End If
```

Optionally, you can also specify a time to wait for the lock. The method will wait the specified amount of time to acquire the lock. If a lock is acquired before the time expires, the method returns *true*; otherwise it returns *false*. The following call waits for one second before giving up on acquiring the lock.

**C#**

```
System.Threading.Monitor.TryEnter(locking object, TimeSpan.FromSeconds(1))
```

**Visual Basic**

```
System.Threading.Monitor.TryEnter(locking object, TimeSpan.FromSeconds(1))
```

One advantage of using *TryEnter* versus *Enter* is that you can potentially recover from deadlock scenarios. Your code won't deadlock if all your lock statements waited only 100 ms to acquire a lock. However, you now have to figure out what to do when you do *not* acquire a lock and also make sure that 100 ms is long enough to wait on a lock. For example, if it takes one second to process a list and a thread waits one second to acquire the list, you may find that acquiring the lock for the list randomly times out. This could be a result of the processor being overwhelmed, the operating system not getting back to your thread in time, and so on. For this reason, you should base your timeout on how long is too long to wait—not on how long it should take for an operation to complete.



**Tip** To help with timeouts, you can reduce the wait duration and make multiple access attempts. For example, instead of waiting 10 seconds, you could wait 500 ms for up to 20 times. This approach allows you to do other things while you wait, such as check to see if you should abort, notify users of application progress, and notify the thread that is using the collection that you are waiting.

For the collection classes, the object you pass as the locking variable is *ICollection.SyncRoot* and not the collection instance itself. You use the *ICollection.SyncRoot* property for synchronizing the collection portion of an object. However, that's not the case with the actual collection instance itself.

Here's an example showing incorrect use.

#### C#

```
ArrayEx<int> list = new ArrayEx<int>();

lock (list)
{
    // Critical code section
}
```

#### Visual Basic

```
Dim list As ArrayEx(Of Integer) = New ArrayEx(Of Integer)
SyncLock list
    ' Critical code section
End SyncLock
```

Here's a correct example.

#### C#

```
ArrayEx<int> list = new ArrayEx<int>();

lock (((ICollection)list).SyncRoot)
{
    // Critical code section
}
```

### Visual Basic

```
Dim list As ArrayEx(Of Integer) = New ArrayEx(Of Integer)
SyncLock CType(list, ICollection).SyncRoot
    ' Critical code section
End SyncLock
```

Using *ICollection.SyncRoot* guarantees that you are synchronizing on a collection operation, whereas using *list* means you are synchronizing on whatever object everyone else using the collection is synchronizing on. This could result in a deadlock, because you do not know what they may be synchronizing on.

## Using the *ReaderWriterLockSlim* Class

The problem with using the *Monitor* class with collections is that it doesn't allow more than one object to lock it at a time. With a collection, you can have more than one thread read the collection at a time, but only one thread write to the collection at a time. Look again at the example in the "Why Should I Care About Thread Synchronization?" section. The section implemented *SumArray* and *ReverseArray* methods. The *SumArray* method represents a reader thread, whereas the *ReverseArray* method represents a writer thread. You could introduce another method called *AverageArray*, which simulates another reader thread as follows.

### C#

```
static void AverageArray(object data)
{
    for ( ; ; )
    {
        State state = (State)data;

        int sum = 0;
        float average = 0;
        Console.WriteLine("Averaging array");
        for (int i = 0; i < state.Items.Count; ++i)
        {
            sum += state.Items[i];
        }
        average = (float)sum / (float)state.Items.Count;
        if (state.Average != average)
        {
            Console.WriteLine("Calculated average was {0} instead of {1}",
                average, state.Average);
        }
        else
        {
            Console.WriteLine("Average is {0}", average);
        }
        Thread.Sleep(100);
    }
}
```

### Visual Basic

```
Sub AverageArray(ByVal data As Object)
    While (True)
        Dim state As State = CType(data, State)

        Dim sum As Integer = 0
        Dim average As Single = 0
        Console.WriteLine("Averaging array")

        For i As Integer = 0 To state.Items.Count - 1
            sum += state.Items(i)
        Next
        average = CSng(sum) / CSng(state.Items.Count)
        If (state.Average <> average) Then
            Console.WriteLine("Calculated average was {0} instead of {1}",
                average, state.Average)
        Else
            Console.WriteLine("Average is {0}", average)
        End If
        Thread.Sleep(100)
    End While
End Sub
```

The *AverageArray* method simply calculates the mean average of the values in the array and displays the resulting value on the screen, whether the average is calculated correctly or not.

You'll also need to modify the *SumArray* and *ReverseArray* methods to handle a structure for the argument instead of an *ArrayEx(T)*. (The code in bold represents the changed code.)

### C#

```
static void ReverseArray(object data)
{
    for ( ; ; )
    {
        State state = (State) data;

        Console.WriteLine("Reversing array");
        for (int i = 0; i < state.Items.Count / 2; ++i)
        {
            int tmp = state.Items[i];
            state.Items[i] = state.Items[state.Items.Count - 1 - i];
            state.Items[state.Items.Count - 1 - i] = tmp;
        }
        Console.WriteLine("Reversed array");
        Thread.Sleep(100);
    }
}

static void SumArray(object data)
```

```

{
    for ( ; ; )
    {
        State state = (State)data;

        int sum = 0;
        Console.WriteLine("Summing array");
        for (int i = 0; i < state.Items.Count; ++i)
        {
            sum += state.Items[i];
        }
        if (state.Sum != sum)
        {
            Console.WriteLine("Calculated sum was {0} instead of {1}", sum, state.Sum);
        }
        else
        {
            Console.WriteLine("Sum is {0}", sum);
        }
        Thread.Sleep(100);
    }
}

```

### Visual Basic

```

Sub ReverseArray(ByVal data As Object)
    While (True)
        Dim state As State = CType(data, State)

        Console.WriteLine("Reversing array")
        For i As Integer = 0 To (state.Items.Count / 2) - 1
            Dim tmp As Integer = state.Items(i)
            state.Items(i) = state.Items(state.Items.Count - 1 - i)
            state.Items(state.Items.Count - 1 - i) = tmp
        Next
        Console.WriteLine("Reversed array")
        Thread.Sleep(100)
    End While
End Sub

Sub SumArray(ByVal data As Object)
    While (True)
        Dim state As State = CType(data, State)

        Dim sum As Integer = 0
        Console.WriteLine("Summing array")
        For i As Integer = 0 To state.Items.Count - 1
            sum += state.Items(i)
        Next
        If (state.Sum <> sum) Then
            Console.WriteLine("Calculated sum was {0} instead of {1}", sum, state.Sum)
        Else
            Console.WriteLine("Sum is {0}", sum)
        End If
        Thread.Sleep(100)
    End While
End Sub

```

The *State* class is as follows.

**C#**

```
public class State
{
    public ArrayEx<int> Items;
    public ReaderWriterLockSlim Lock;
    public int Sum;
    public float Average;
}
```

**Visual Basic**

```
Public Class State
    Public Items As ArrayEx(Of Integer)
    Public Lock As ReaderWriterLockSlim
    Public Sum As Integer
    Public Average As Single
End Class
```

The *State* class holds the items used in the calculation as well as a *Sum* and *Average* of the items. The *Lock* field will not be used yet.

The following code drives the sample.

**C#**

```
State data = new State();
data.Lock = new ReaderWriterLockSlim();
data.Items = new ArrayEx<int>();
data.Sum = 0;
for (int i = 0; i < 10000; ++i)
{
    data.Items.Add(i);
    data.Sum += i;
}
data.Average = (float)data.Sum / (float)data.Items.Count;

ThreadPool.QueueUserWorkItem(new WaitCallback(ReverseArray), data);
ThreadPool.QueueUserWorkItem(new WaitCallback(SumArray), data);
ThreadPool.QueueUserWorkItem(new WaitCallback(AverageArray), data);

Thread.Sleep(3000);
```

**Visual Basic**

```
Dim data As State = New State()
data.Lock = New ReaderWriterLockSlim()
data.Items = New ArrayEx(Of Integer)
data.Sum = 0
For i As Integer = 0 To 9
    data.Items.Add(i)
    data.Sum += i
Next
data.Average = CSng(data.Sum) / CSng(data.Items.Count)
```

```
ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf ReverseArray), data)
ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf SumArray), data)
ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf AverageArray), data)

Thread.Sleep(3000)
```

### Output

```
Reversing array
Averaging array
Summing array
Calculated average was 3557 instead of 4999
Calculated sum was 32600504 instead of 49995000
Reversed array
Averaging array
Summing array
Reversing array
Calculated average was 5438 instead of 4999
Calculated sum was 61774504 instead of 49995000
Reversed array
Summing array
Averaging array
Reversing array
Calculated sum was 48314241 instead of 49995000
Calculated average was 4211 instead of 4999
Reversed array
Summing array
Reversing array
Averaging array
Calculated sum was 63563839 instead of 49995000
Calculated average was 5400 instead of 4999
Reversed array
```

As you can see, without synchronization support, this code still has synchronization problems. You can use the *ReaderWriterLockSlim* class to allow the two reader threads access to the collection when the writer thread isn't writing to the collection, and vice versa.

The reader threads will need to use the *EnterReadLock* and *ExitReadLock* methods. The *EnterReadLock* method attempts to acquire the lock as a reader. When it succeeds, writer threads will not be able to acquire the lock until no reader thread still has the lock. Reader threads must call *ExitReadLock* to release the lock when they no longer need it. The following code shows the *AverageArray* method modified to support synchronization using *ReaderWriterLockSlim*.

### C#

```
static void AverageArray(object data)
{
    for (; ; )
    {
        State state = (State)data;

        int sum = 0;
        float average = 0;
```

```
Console.WriteLine("Averaging array");
state.Lock.EnterReadLock();
try
{
    for (int i = 0; i < state.Items.Count; ++i)
    {
        sum += state.Items[i];
    }
    average = (float)sum / (float)state.Items.Count;
}
finally
{
    state.Lock.ExitReadLock();
}
if (state.Average != average)
{
    Console.WriteLine("Calculated average was {0} instead of {1}",
                      average, state.Average);
}
else
{
    Console.WriteLine("Average is {0}", average);
}
Thread.Sleep(100);
}
```

## Visual Basic

```
Sub AverageArray(ByVal data As Object)
    While (True)
        Dim state As State = CType(data, State)

        Dim sum As Integer = 0
        Dim average As Single = 0
        Console.WriteLine("Averaging array")
        state.Lock.EnterReadLock()

        Try
            For i As Integer = 0 To state.Items.Count - 1
                sum += state.Items(i)
            Next
            average = CSng(sum) / CSng(state.Items.Count)
        Finally
            state.Lock.ExitReadLock()
        End Try
        If (state.Average <> average) Then
            Console.WriteLine("Calculated average was {0} instead of {1}", _
                              average, state.Average)
        Else
            Console.WriteLine("Average is {0}", average)
        End If
        Thread.Sleep(100)
    End While
End Sub
```

Note how the *ExitReadLock* method gets called within the *finally* statement. This ensures that the lock gets released even if the code throws an exception.

The *EnterReadLock* method does not return until it acquires the lock. You will need to use the *TryEnterReadLock* method instead if you want to return when the lock has already been acquired by a writer thread. The following code is the modified *SumArray* method that uses *TryEnterReadLock* instead of *EnterReadLock*.

C#

```
static void SumArray(object data)
{
    for ( ; ; )
    {
        State state = (State)data;

        int sum = 0;
        Console.WriteLine("Summing array");
        if (state.Lock.TryEnterReadLock(TimeSpan.FromMilliseconds(100)))
        {
            if (state.Lock.CurrentReadCount > 1)
            {
                Console.WriteLine("Sharing the lock with AverageArray");
            }
            try
            {
                for (int i = 0; i < state.Items.Count; ++i)
                {
                    sum += state.Items[i];
                }
            }
            finally
            {
                state.Lock.ExitReadLock();
            }
            if (state.Sum != sum)
            {
                Console.WriteLine("Calculated sum was {0} instead of {1}", sum, state.Sum);
            }
            else
            {
                Console.WriteLine("Sum is {0}", sum);
            }
        }
        else
        {
            Console.WriteLine("Couldn't acquire lock");
        }
        Thread.Sleep(100);
    }
}
```

## Visual Basic

```
Sub SumArray(ByVal data As Object)
    While (True)
        Dim state As State = CType(data, State)

        Dim sum As Integer = 0
        Console.WriteLine("Summing array")
        If (state.Lock.TryEnterReadLock(TimeSpan.FromMilliseconds(100))) Then
            If (state.Lock.CurrentReadCount > 1) Then
                Console.WriteLine("Sharing the lock with AverageArray")
            End If
            Try

                For i As Integer = 0 To state.Items.Count - 1
                    sum += state.Items(i)
                Next
                Finally
                    state.Lock.ExitReadLock()
                End Try

                If (state.Sum <> sum) Then
                    Console.WriteLine("Calculated sum was {0} instead of {1}", sum, state.Sum)
                Else
                    Console.WriteLine("Sum is {0}", sum)
                End If
            Else
                Console.WriteLine("Couldn't acquire lock")
            End If
            Thread.Sleep(100)
        End While
    End Sub
```

The code will print "Couldn't acquire lock" if *TryEnterReadLock* fails to acquire the lock. If it succeeds, the code is the same as *EnterReadLock*. The *SumArray* method also uses the *CurrentReadCount* property to display whether it is sharing the lock with the *AverageArray* method. The *CurrentReadCount* property returns the number of read threads that have currently acquired the lock.

The writer thread needs to use the *EnterWriteLock* and *ExitWriteLock* methods. The *EnterWriteLock* method attempts to acquire the lock as a writer. When it succeeds, other reader or writer threads will not be able to acquire the lock until it is released. The writer thread must call *ExitWriterLock* to release the lock when it is no longer needed. The following shows the *ReverseArray* method modified to support synchronization using *ReaderWriterLockSlim*.

## C#

```

static void ReverseArray(object data)
{
    for ( ; ; )
    {
        State state = (State)data;

        Console.WriteLine("Reversing array");
        state.Lock.EnterWriteLock();
        try
        {
            for (int i = 0; i < state.Items.Count / 2; ++i)
            {
                int tmp = state.Items[i];
                state.Items[i] = state.Items[state.Items.Count - 1 - i];
                state.Items[state.Items.Count - 1 - i] = tmp;
            }
        }
        finally
        {
            state.Lock.ExitWriteLock();
        }
        Console.WriteLine("Reversed array");
        Thread.Sleep(100);
    }
}

```

## Visual Basic

```

Sub ReverseArray(ByVal data As Object)
    While (True)
        Dim state As State = CType(data, State)

        Console.WriteLine("Reversing array")
        state.Lock.EnterWriteLock()
        Try

            For i As Integer = 0 To (state.Items.Count / 2) - 1
                Dim tmp As Integer = state.Items(i)
                state.Items(i) = state.Items(state.Items.Count - 1 - i)
                state.Items(state.Items.Count - 1 - i) = tmp
            Next
        Finally
            state.Lock.ExitWriteLock()
        End Try
        Console.WriteLine("Reversed array")
        Thread.Sleep(100)
    End While
End Sub

```

Again, the *ExitWriteLock* occurs within the *finally* statement to ensure that the lock is released even if the code throws an exception. You could use the *TryEnterWriteLock* method to avoid blocking while waiting for the writer lock.

Here's the output when the preceding example executes.

### Output

```
Reversing array
Averaging array
Summing array
Reversed array
Sharing the lock with AverageArray
Average is 4999.5
Sum is 49995000
Reversing array
Reversed array
Averaging array
Summing array
Sharing the lock with AverageArray
Average is 4999.5
Sum is 49995000
Reversing array
Reversed array
Summing array
Averaging array
Sum is 49995000
Average is 4999.5
Reversing array
Reversed array
Averaging array
Summing array
Sharing the lock with AverageArray
Average is 4999.5
Sum is 49995000
Reversing array
Reversed array
Summing array
Averaging array
Sum is 49995000
Average is 4999.5
Reversing array
Reversed array
Averaging array
Summing array
Sum is 49995000
Average is 4999.5
Reversing array
Reversed array
Averaging array
Summing array
Average is 4999.5
Sum is 49995000
Reversing array
Reversed array
```

Notice that the synchronization problems are no longer present. You can also see the random nature of threads and sleeps by looking at the unpredictable pattern of the output. Sometimes the lock is shared with the *AverageArray* thread; sometimes it isn't.

## Handling Recursion

In the previous example, the threads execute only within one method, but in real-world programming, you will also call methods from threads that interact with a collection instead of simply having the thread interact directly with the collection. These methods could, in turn, acquire the lock and then potentially call other methods or events that do the same, in a process referred to as *recursive locking*.

There are times when you will want to allow recursive locking and times when you will not. With the *ReaderWriterLockSlim* class, you can define the recursive locking policy you want when you first create the lock. The two options are *LockRecursionPolicy.NoRecursion* and *LockRecursionPolicy.SupportRecursion*. You are not able to call *EnterReadLock*, *EnterWriteLock*, or *EnterUpgradeableReadLock* within another *EnterReadLock*, *EnterWriteLock*, or *EnterUpgradeableReadLock* if *LockRecursionPolicy.NoRecursion* is set.

When *LockRecursionPolicy.SupportRecursion* is set, you can enter a lock from within another lock. Each call to *EnterXLock* or *TryEnterXLock* must call the corresponding *ExitXLock*; however, you do not have to call *ExitXLock* in the exact reverse order that you called the *EnterXLock* calls. You can use the *IsReadLockHeld*, *IsUpgradeableReadLockHeld*, and *IsWriteLockHeld* properties to determine whether the current thread has already called *EnterReadLock*, *EnterWriteLock*, or *EnterUpgradeableReadLock*.

## Using Upgradeable Locks

Sometimes you may find that a thread needs to write data only when a certain condition is met, but needs to read data to determine whether it should acquire a write mode. For example, suppose you have a collection of events that you want to remove and execute. The class is defined as the following.

### C#

```
public class ScheduledEvent
{
    public DateTime ExecutionTime;
    public string Name;
    public Action Action;
}
```

### Visual Basic

```
Public Class ScheduledEvent
    Public ExecutionTime As DateTime
    Public Name As String
    Public Action As Action
End Class
```

The *ExecutionTime* is the time at which the event should occur, and *Action* is the action that should take place at the specified time. The list and lock are defined as the following.

### C#

```
ReaderWriterLockSlim lckEvents = new ReaderWriterLockSlim();
ArrayEx<ScheduledEvent> events = new ArrayEx<ScheduledEvent>();
```

### Visual Basic

```
Dim lckEvents As ReaderWriterLockSlim = New ReaderWriterLockSlim()
Dim events As ArrayEx(Of ScheduledEvent) = New ArrayEx(Of ScheduledEvent)()
```

More than one thread may be receiving the events. Perhaps one thread adds to the events, one removes events from the list and executes them, and yet another thread displays the list to users when requested—and maybe to other threads that read from the list. The code for the execution and removal thread might look like the following.

### C#

```
for (; ; )
{
    lckEvents.EnterWriteLock();
    try
    {
        for (int i = 0; i < events.Count; )
        {
            if (events[i].ExecutionTime <= DateTime.Now)
            {
                events[i].Action();

                // Remove the item from the list
                events.RemoveAt(i);
            }
            else
            {
                ++i;
            }
        }
    }
    finally
    {
        lckEvents.ExitWriteLock();
    }
    Thread.Sleep(100);
}
```

**Visual Basic**

```
While (True)
    lckEvents.EnterWriteLock()
    Try
        For i As Integer = 0 To events.Count - 1
            If (events(i).ExecutionTime <= DateTime.Now) Then
                events(i).Action()

                    ' Remove the item from the list
                events.RemoveAt(i)
            Else
                i += 1
            End If
        Next
    Finally
        lckEvents.ExitWriteLock()
    End Try
    Thread.Sleep(100)
End While
```

The downside of doing it this way is that no other threads may access the list even if the collection isn't modified. You can use the upgradeable locks to lock a collection for read-only access and change the lock to write access later. Take a look at the following example.

**C#**

```
for (; ; )
{
    bool enterWriteLock = false;
    lckEvents.EnterUpgradeableReadLock();
    try
    {
        for (int i = 0; i < events.Count; )
        {
            if (events[i].ExecutionTime <= DateTime.Now)
            {

                if (!enterWriteLock)
                {
                    lckEvents.EnterWriteLock();
                    enterWriteLock = true;
                }

                events[i].Action();

                // Remove the item from the list
                events.RemoveAt(i);
            }
            else
```

```
        {
            ++
        }
    }
finally
{
    if (enterWriteLock)
    {
        lckEvents.ExitWriteLock();
        enterWriteLock = false;
    }
    lckEvents.ExitUpgradeableReadLock();
}
Thread.Sleep(100);
}
```

### Visual Basic

```
While (True)
    Dim enterWriteLock As Boolean = False
    lckEvents.EnterUpgradeableReadLock()
    Try
        For i As Integer = 0 To events.Count - 1
            If (events(i).ExecutionTime <= DateTime.Now) Then

                If (Not enterWriteLock) Then
                    lckEvents.EnterWriteLock()
                    enterWriteLock = True
                End If

                events(i).Action()

                ' Remove the item from the list
                events.RemoveAt(i)

            Else
                i += 1
            End If
        Next
    Finally
        If (enterWriteLock) Then
            lckEvents.ExitWriteLock()
            enterWriteLock = False
        End If
        lckEvents.ExitUpgradeableReadLock()
    End Try
    Thread.Sleep(100)
End While
```

The thread now acquires an *UpgradeableReadLock*. If this finds an event that should be executed, a write lock is then acquired. When an upgradeable lock has been acquired, any number of threads can acquire a read lock as long as no threads are waiting to acquire a write lock. Only one thread may acquire an upgradeable lock at a time.

## Implementing a Synchronized Wrapper Class

A Synchronized Wrapper class wraps a collection class that doesn't have *IsSynchronized* set to *true*, although it could also be used for one that does have *IsSynchronized* set to *true*. This allows you to return a collection object that isn't synchronized to the user in an object that is. It also gives a common synchronized implementation and eliminates some of the extra code written to synchronize. This may be useful if your code is accessed through a script, an API, or reflection.

The following shows a simple way to provide a Synchronized Wrapper around a class that implements *IList(T)*.

C#

```
public class SynchronizedListWrapper<T> : IList<T>, ICollection<T>
{
    object m_syncRoot;
    IList<T> m_items;

    public SynchronizedListWrapper(IList<T> items)
    {
        if (items == null)
        {
            throw new ArgumentNullException("items");
        }
        m_items = items;
        m_syncRoot = new object();
    }

    public SynchronizedListWrapper(IList<T> items, object syncRoot)
    {
        if (items == null)
        {
            throw new ArgumentNullException("items");
        }
        if (syncRoot == null)
        {
            throw new ArgumentNullException("syncRoot");
        }
        m_items = items;
        m_syncRoot = syncRoot;
    }

    #region IList<T> Members

    public int IndexOf(T item)
    {
        lock (m_syncRoot)
        {
            return m_items.IndexOf(item);
        }
    }

}
```

```
public void Insert(int index, T item)
{
    lock (m_syncRoot)
    {
        m_items.Insert(index, item);
    }
}

public void RemoveAt(int index)
{
    lock (m_syncRoot)
    {
        m_items.RemoveAt(index);
    }
}

public T this[int index]
{
    get
    {
        lock (m_syncRoot)
        {
            return m_items[index];
        }
    }
    set
    {
        lock (m_syncRoot)
        {
            m_items[index] = value;
        }
    }
}

#endregion

#region ICollection<T> Members

public void Add(T item)
{
    lock (m_syncRoot)
    {
        m_items.Add(item);
    }
}

public void Clear()
{
    lock (m_syncRoot)
    {
        m_items.Clear();
    }
}
```

```
public bool Contains(T item)
{
    lock (_syncRoot)
    {
        return _items.Contains(item);
    }
}

public void CopyTo(T[] array, int arrayIndex)
{
    lock (_syncRoot)
    {
        _items.CopyTo(array, arrayIndex);
    }
}

public int Count
{
    get
    {
        lock (_syncRoot)
        {
            return _items.Count;
        }
    }
}

public bool IsReadOnly
{
    get
    {
        return _items.IsReadOnly;
    }
}

public bool Remove(T item)
{
    lock (_syncRoot)
    {
        return _items.Remove(item);
    }
}

#endregion

#region IEnumerable<T> Members

public IEnumerator<T> GetEnumerator()
{
    return _items.GetEnumerator();
}


```

```
#endregion

#region IEnumerable Members

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return m_items.GetEnumerator();
}

#endregion

#region ICollection Members

public void CopyTo(Array array, int index)
{
    lock (m_syncRoot)
    {
        Array.Copy(m_items.ToArray(), 0, array, index, Count);
    }
}

public bool IsSynchronized
{
    get { return true; }
}

public object SyncRoot
{
    get { return m_syncRoot; }
}

#endregion
}
```

### Visual Basic

```
Public Class SynchronizedListWrapper(Of T)
    Implements IList(Of T)
    Implements ICollection

    Private m_syncRoot As Object
    Private m_items As IList(Of T)

    Public Sub New(ByVal items As IList(Of T))
        If (items Is Nothing) Then
            Throw New ArgumentNullException("items")
        End If
        m_items = items
        m_syncRoot = New Object()
    End Sub
```

```
Public Sub New(ByVal items As IList(Of T), ByVal syncRoot As Object)
    If (items Is Nothing) Then
        Throw New ArgumentNullException("items")
    End If
    If (syncRoot Is Nothing) Then
        Throw New ArgumentNullException("syncRoot")
    End If
    m_items = items
    m_syncRoot = syncRoot
End Sub

#Region "IList(T) Members"

Public Function IndexOf(ByVal item As T) As Integer Implements IList(Of T).IndexOf
    SyncLock m_syncRoot
        Return m_items.IndexOf(item)
    End SyncLock
End Function

Public Sub Insert(ByVal index As Integer, ByVal item As T) Implements IList(Of T).Insert
    SyncLock m_syncRoot
        m_items.Insert(index, item)
    End SyncLock
End Sub

Public Sub RemoveAt(ByVal index As Integer) Implements IList(Of T).RemoveAt
    SyncLock m_syncRoot
        m_items.RemoveAt(index)
    End SyncLock
End Sub

Default Public Property Item(ByVal index As Integer) As T Implements IList(Of T).Item
    Get
        SyncLock m_syncRoot
            Return m_items(index)
        End SyncLock
    End Get
    Set(ByVal value As T)
        SyncLock m_syncRoot
            m_items(index) = value
        End SyncLock
    End Set
End Property

#End Region

#Region "ICollection(T) Members"

Public Sub Add(ByVal item As T) Implements ICollection(Of T).Add
    SyncLock m_syncRoot
        m_items.Add(item)
    End SyncLock
End Sub
```

```
Public Sub Clear() Implements ICollection(Of T).Clear
    SyncLock m_syncRoot
        m_items.Clear()
    End SyncLock
End Sub

Public Function Contains(ByVal item As T) As Boolean _
    Implements ICollection(Of T).Contains
    SyncLock m_syncRoot
        Return m_items.Contains(item)
    End SyncLock
End Function

Public Sub CopyTo(ByVal array As T(), ByVal arrayIndex As Integer) _
    Implements ICollection(Of T).CopyTo
    SyncLock m_syncRoot
        m_items.CopyTo(array, arrayIndex)
    End SyncLock
End Sub

Public ReadOnly Property Count() As Integer _
    Implements ICollection(Of T).Count, ICollection.Count
    Get
        SyncLock m_syncRoot
            Return m_items.Count
        End SyncLock
    End Get
End Property

Public ReadOnly Property IsReadOnly() As Boolean _
    Implements ICollection(Of T).IsReadOnly
    Get
        SyncLock m_syncRoot
            Return m_items.IsReadOnly
        End SyncLock
    End Get
End Property

Public Function Remove(ByVal item As T) As Boolean _
    Implements ICollection(Of T).Remove
    SyncLock m_syncRoot
        Return m_items.Remove(item)
    End SyncLock
End Function

#End Region

#Region "IEnumerable<T> Members"

Public Function GetEnumerator() As IEnumerator(Of T) _
    Implements IEnumerable(Of T).GetEnumerator
    Return m_items.GetEnumerator()
End Function

#End Region
```

```

#Region "IEnumerable Members"

    Private Function GetEnumerator2() As System.Collections.IEnumerator _
        Implements IEnumerable.GetEnumerator
        Return m_items.GetEnumerator()
    End Function

#End Region

#Region "ICollection Members"

    Private Sub CopyTo2(ByVal array As Array, ByVal index As Integer) _
        Implements ICollection.CopyTo
        SyncLock m_syncRoot
            array.Copy(m_items.ToArray(), 0, array, index, Count)
        End SyncLock
    End Sub

    Private ReadOnly Property IsSynchronized() As Boolean _
        Implements ICollection.IsSynchronized
        Get
            Return True
        End Get
    End Property

    Private ReadOnly Property SyncRoot() As Object Implements ICollection.SyncRoot
        Get
            Return m_syncRoot
        End Get
    End Property

#End Region

End Class

```

The constructor accepts the *IList(T)* you are providing synchronization support to and an optional *SyncRoot* that you want to use for synchronization. You can provide synchronization support to the *ArrayEx(T)* class by doing the following.

#### C#

```

ArrayEx<int> list = new ArrayEx<int>() { 55, 8 };

SynchronizedListWrapper<int> slist = new SynchronizedListWrapper<int>(list);

```

#### Visual Basic

```

Dim list As New ArrayEx(Of Integer)(New Integer() {55, 8})
Dim slist As New SynchronizedReadOnlyCollection(Of Integer)(list)

```

Now all the calls you make to *list* using the *slist* variable are synchronized. You could also later do the following.

**C#**

```
lock(((System.Collections.ICollection)slist).SyncRoot)
{
    list.Add(12);
}
```

**Visual Basic**

```
SyncLock (CType(list, System.Collections.ICollection).SyncRoot)
    list.Add(12)
End SyncLock
```

A 12 is added to the collection list using the synchronization object in *slist*.

## Handling Collection Changes While Enumerating

Enumerating over a collection while it is changing can cause some undesirable exceptions such as memory access violations, null pointer exceptions, and out-of-range exceptions. Luckily, the collection classes you implemented in the first three chapters of this book contain the field *m\_updateCode* to help warn the user that the collection being iterated over has changed. The enumerators in Chapter 6 store the value of *m\_updateCode* before enumerating begins. The value is then checked whenever *IEnumerator.MoveNext* is called to verify that the collection being enumerated has not changed. An exception is thrown if the value has changed from the initial value. This gives a warning to the user that the collection being enumerated over has changed.

## Synchronized Collection Classes

The .NET Framework defines several thread-safe, type-safe classes for working with collections. They are the *SynchronizedCollection(T)*, *SynchronizedKeyedCollection(T)*, and *SynchronizedReadOnlyCollection(T)* classes.

### *SynchronizedCollection(T)*

The *SynchronizedCollection(T)* class provides synchronization support for an array. All of the items are stored in a *List(T)* class that may be accessed through the *Items* property. You use the *SynchronizedCollection(T)* class in the same way that you use the *List(T)* class except the methods and properties are thread safe. You can pass the *SyncRoot* object in as a parameter to the constructor. This is useful if you want several collections or objects to be synchronized using the same object.

## ***SynchronizedKeyedCollection(T)***

The *SynchronizedKeyedCollection(T)* class provides synchronization support for an associative array. All items are stored in a *Dictionary(T)* class, but the class doesn't provide a way to access the *Dictionary(T)* object. You use *SynchronizedKeyedCollection(T)* in the same way that you use the *Dictionary(T)* class except that the methods and properties are thread safe. You can pass the *SyncRoot* object in as a parameter to the constructor. This is useful if you want several collections or objects to be synchronized using the same object.

## ***SynchronizedReadOnlyCollection(T)***

The *SynchronizedReadOnlyCollection(T)* class provides synchronization support to a read-only collection. This class allows you to perform synchronized read-only methods and properties on the collection. You can pass the *SyncRoot* object in as a parameter to the constructor. This is useful if you want several collections or objects to be synchronized using the same object.

## **Summary**

In this chapter, you learned about threads and how to use them with collections. With threads, you can do simultaneous operations on collections that require you to synchronize access to your collections. You also learned about objects that may be used to synchronize your collections and about some built-in synchronized collection classes for working with collections that are not synchronized.

# Chapter 9

# Serializing Collections

After completing this chapter, you will be able to

- Understand what serialization is.
- Use the serialization formatters.
- Add serialization support to your custom collection classes.

## Serialization

*Serialization* is the process of converting an object in a defined way to a format that can be stored and retrieved. The formatted data can be stored in memory, in a file, in a database, through a data stream sent over the network, and in other ways. The storage format could be a plain text format, such as XML, or a pure binary format.

## Using the *Serializer* Formatters

The formatter classes are one of the many ways to serialize data. Classes can also be serialized using *XmlSerializer*. The following examples use the formatter classes, which let you choose among binary, custom serializers, and more.

## Applying the *Serializable* Attribute

The *Serializable* attribute defines a class as being serializable. All fields in the class or structure are serialized unless the *NonSerializable* attribute is applied to them.



**Note** The *Serializable* and *NonSerializable* attributes are actually named as the *SerializableAttribute* and *NonSerializableAttribute* classes. The compiler lets you remove the postfix *Attribute* from the names when using them as attributes in your code.

Take a look at the following example.

**C#**

```
[Serializable]
class MyClass
{
    public string PublicField;
    private string PrivateField;
    protected string ProtectedField;

    [NonSerialized()]
    public string NonSerializedField;

    public string Property
    {
        get { return NonSerializedField; }
        set { NonSerializedField = value; }
    }

    public string PrivateFieldValue
    {
        get { return PrivateField; }
    }

    public string ProtectedFieldValue
    {
        get { return ProtectedField; }
    }

    public MyClass(string publicField, string privateField, string protectedField,
                  string nonSerializedField)
    {
        PublicField = publicField;
        PrivateField = privateField;
        ProtectedField = protectedField;
        NonSerializedField = nonSerializedField;
    }
}
```

**Visual Basic**

```
<Serializable()> _
Public Class [MyClass]
    Public PublicField As String
    Private PrivateField As String
    Protected ProtectedField As String

    <NonSerialized()> _
    Public NonSerializedField As String

    Public Property [Property]() As String
        Get
            Return NonSerializedField
        End Get
    End Property
```

```
Set(ByVal value As String)
    NonSerializedField = value
End Set
End Property

Public ReadOnly Property PrivateFieldValue() As String
Get
    Return PrivateField
End Get
End Property

Public ReadOnly Property ProtectedFieldValue() As String
Get
    Return protectedfield
End Get
End Property

Sub New(ByVal publicField As String, ByVal privateField As String, _
        ByVal protectedField As String, ByVal nonSerializedField As String)
    Me.PublicField = publicField
    Me.PrivateField = privateField
    Me.ProtectedField = protectedField
    Me.NonSerializedField = nonSerializedField
End Sub
End Class
```

The preceding class is marked as serializable by using the *Serializable* attribute. Four fields are defined along with a public property. The *PublicField* is declared public, the *PrivateField* as private, and the *ProtectedField* as protected. The last field, *NonSerializableField*, is defined as public and has the *NonSerializable* attribute set. A public property is also defined so that you can see what happens with properties. The following code is used to serialize the data to memory and then deserialize it from memory.

### C#

```
MemoryStream stream = new MemoryStream();
BinaryFormatter formatter = new BinaryFormatter();

MyClass data = new MyClass("A", "B", "C", "D");

formatter.Serialize(stream, data);

stream.Position = 0;

MyClass deserialized = (MyClass)formatter.Deserialize(stream);

Console.WriteLine("PublicField={0}", deserialized.PublicField);
Console.WriteLine("Property={0}", deserialized.Property);
Console.WriteLine("NonSerializedField={0}", deserialized.NonSerializedField);
Console.WriteLine("PrivateField={0}", deserialized.PrivateFieldValue);
Console.WriteLine("ProtectedField={0}", deserialized.ProtectedFieldValue);
```

### Visual Basic

```
Dim stream As MemoryStream = New MemoryStream()
Dim formatter As BinaryFormatter = New BinaryFormatter()

Dim data As [MyClass] = New [MyClass]("A", "B", "C", "D")

formatter.Serialize(stream, data)

stream.Position = 0

Dim deserialized As [MyClass] = CType(formatter.Deserialize(stream), [MyClass])

Console.WriteLine("PublicField={0}", deserialized.PublicField)
Console.WriteLine("Property={0}", deserialized.Property)
Console.WriteLine("NonSerializedField={0}", deserialized.NonSerializedField)
Console.WriteLine("PrivateField={0}", deserialized.PrivateFieldValue)
Console.WriteLine("ProtectedField={0}", deserialized.ProtectedFieldValue)
```

Serializing requires a stream and a formatter. You can use any class that derives from the *Stream* class to stream the contents, and you can use any class that implements the *IFormatter* interface as a formatter. The base *Formatter* class is intended to help simplify the creation of custom formatter classes when you need to create your own. The *IFormatter* interface contains two methods: *Serialize* and *Deserialize*. The *Serialize* method serializes the specified data into the specified stream; and the *Deserialize* method, as you may have guessed, deserializes data from a specified stream, recreating the original objects. The preceding example uses a *MemoryStream* for serialization. The code moves the *MemoryStream* position back to the beginning after the data has been serialized so that the *Deserialize* method will start from the beginning of the stream. The example then displays the results of the deserialization on the screen as follows.

### Output

```
PublicField=A
Property=
NonSerializedField=
PrivateField=B
ProtectedField=C
```

Notice how all fields are serialized except the *NonSerializedField* field. This is because that field had the *NonSerializable* attribute set on it. Properties are not serialized either. So, the *NonSerializedField* field was not serialized using the *Property* property either. Also, calling *serialize* on an object that has not been decorated with the *Serializable* attribute will result in an exception, as shown in the following code.

### C#

```
struct NonSerializableStruct
{
    public double X;
    public double Y;
}
```

### Visual Basic

```
Structure NonSerializableStruct
    Public X As Double
    Public Y As Double
End Structure
```

The following code attempts to serialize the preceding structure.

### C#

```
NonSerializableStruct data = new NonSerializableStruct();

MemoryStream stream = new MemoryStream();
BinaryFormatter formatter = new BinaryFormatter();

try
{
    formatter.Serialize(stream, data);
    Console.WriteLine("Data was serialized");
}
catch (SerializationException ex)
{
    Console.WriteLine("Class isn't marked as serializable");
}
```

### Visual Basic

```
Dim data As NonSerializableStruct = New NonSerializableStruct()

Dim stream As MemoryStream = New MemoryStream()
Dim formatter As BinaryFormatter = New BinaryFormatter()

Try
    formatter.Serialize(stream, data)
    Console.WriteLine("Data was serialized")
Catch ex As SerializationException
    Console.WriteLine("Class isn't marked as serializable")
End Try
```

### Output

```
Class isn't marked as serializable
```

The *SerializationException* exception is also thrown if the return type of one of the fields does not have the *Serializable* attribute set.

## Controlling Serialization Behavior

As shown in the previous section, all fields are stored unless you apply the *NonSerializable* attribute to them. This may not always be what you want. Sometimes you might find it useful to specify a storage name for the field instead of the default one or control how the data is serialized and deserialized. You can do so with the *ISerializable* interface.

To use the *ISerializable* interface, you need to mark the class as *Serializable*, implement the interface, and add a protected deserialize constructor. Take a look at the following example.

C#

```
[Serializable()]
class MyClass : ISerializable
{
    public string PublicField;
    private string PrivateField;
    protected string ProtectedField;

    [NonSerialized()]
    public string NonSerializedField;

    public string Property
    {
        get { return NonSerializedField; }
        set { NonSerializedField = value; }
    }

    public string PrivateFieldValue
    {
        get { return PrivateField; }
    }

    public string ProtectedFieldValue
    {
        get { return ProtectedField; }
    }

    public MyClass(string publicField, string privateField, string protectedField,
                  string nonSerializedField)
    {
        PublicField = publicField;
        PrivateField = privateField;
        ProtectedField = protectedField;
        NonSerializedField = nonSerializedField;
    }

    #region ISerializable Members

    protected MyClass(SerializationInfo info, StreamingContext context)
    {
        PublicField = info.GetString("PublicField");
        ProtectedField = info.GetString("ProtectedField");
        NonSerializedField = info.GetString("NonSerializedField");
        PrivateField = info.GetString("PrivateField");
    }

    void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
```

```
{  
    info.AddValue("PublicField", PublicField);  
    info.AddValue("ProtectedField", ProtectedField);  
    info.AddValue("NonSerializedField", NonSerializedField);  
    info.AddValue("PrivateField", PrivateField);  
}  
  
#endregion  
}
```

### Visual Basic

```
<Serializable()> _  
Public Class [MyClass]  
    Implements ISerializable  
  
    Public PublicField As String  
    Private PrivateField As String  
    Protected ProtectedField As String  
  
<NonSerialized()>_  
    Public NonSerializedField As String  
  
    Public Property [Property]() As String  
        Get  
            Return NonSerializedField  
        End Get  
        Set(ByVal value As String)  
            NonSerializedField = value  
        End Set  
    End Property  
  
    Public ReadOnly Property PrivateFieldValue() As String  
        Get  
            Return PrivateField  
        End Get  
    End Property  
  
    Public ReadOnly Property ProtectedFieldValue() As String  
        Get  
            Return ProtectedField  
        End Get  
    End Property  
  
    Sub New(ByVal publicField As String, ByVal privateField As String, _  
            ByVal protectedField As String, ByVal nonSerializedField As String)  
        Me.PublicField = publicField  
        Me.PrivateField = privateField  
        Me.ProtectedField = protectedField  
        Me.NonSerializedField = nonSerializedField  
    End Sub
```

```
#Region "ISerializable Members"

    Protected Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
        PublicField = info.GetString("PublicField")
        ProtectedField = info.GetString("ProtectedField")
        NonSerializedField = info.GetString("NonSerializedField")
        PrivateField = info.GetString("PrivateField")
    End Sub

    Sub GetObjectData(ByVal info As SerializationInfo, ByVal context As StreamingContext)
        Implements ISerializable.GetObjectData
        info.AddValue("PublicField", PublicField)
        info.AddValue("ProtectedField", ProtectedField)
        info.AddValue("NonSerializedField", NonSerializedField)
        info.AddValue("PrivateField", PrivateField)
    End Sub

#End Region

End Class
```

The following code can be used to serialize and deserialize the data.

#### C#

```
MemoryStream stream = new MemoryStream();
BinaryFormatter formatter = new BinaryFormatter();

MyClass data = new MyClass("A", "B", "C", "D");

formatter.Serialize(stream, data);

stream.Position = 0;

MyClass deserialized = (MyClass)formatter.Deserialize(stream);

Console.WriteLine("PublicField={0}", deserialized.PublicField);
Console.WriteLine("Property={0}", deserialized.Property);
Console.WriteLine("NonSerializedField={0}", deserialized.NonSerializedField);
Console.WriteLine("PrivateField={0}", deserialized.PrivateFieldValue);
Console.WriteLine("ProtectedField={0}", deserialized.ProtectedFieldValue);
```

#### Visual Basic

```
Dim stream As MemoryStream = New MemoryStream()
Dim formatter As BinaryFormatter = New BinaryFormatter()

Dim data As [MyClass] = New [MyClass]("A", "B", "C", "D")

formatter.Serialize(stream, data)

stream.Position = 0

Dim deserialized As [MyClass] = CType(formatter.Deserialize(stream), [MyClass])
```

```
Console.WriteLine("PublicField={0}", deserialized.PublicField)
Console.WriteLine("Property={0}", deserialized.Property)
Console.WriteLine("NonSerializedField={0}", deserialized.NonSerializedField)
Console.WriteLine("PrivateField={0}", deserialized.PrivateFieldValue)
Console.WriteLine("ProtectedField={0}", deserialized.ProtectedFieldValue)
```

### Output

```
PublicField=A
Property=D
NonSerializedField=D
PrivateField=B
ProtectedField=C
```

As you can see, the *NonSerializable* attribute no longer applies because you stated what you want to serialize and deserialize in the *ISerializable.GetObjectData* method and deserialize constructor.



**Note** One problem with the *Serializable* attribute is that it doesn't handle type changes. For example, if you define a class and serialize the data, the default formatters cannot handle class renames or property renames after the class has been serialized. Although it is possible to inform them of your changes, you might want to look at writing a version number by using the *ISerializable* interface. The deserializing can look at the version number and then handle field name changes.

## Adding Serialization Support to Collection Classes

As you saw in the previous sections, all objects must have the *Serializable* attribute set if they want to be serialized. This means that the objects stored in the collection must be marked as *Serializable* or serializing the object will throw an exception. You could add support for serialization to the custom collection class that you created in earlier chapters by simply adding the *Serializable* attribute to the class and marking some of the field as nonserializable with the *NonSerializable* attribute.

For example, change the *ArrayEx(T)* class definition from the following:

### C#

```
[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(ArrayDebugView))]
public partial class ArrayEx<T>
```

### Visual Basic

```
<DebuggerDisplay("Count={Count}"> _
<DebuggerTypeProxy(GetType(ArrayDebugView))> _
Partial Public Class ArrayEx(Of T)
```

to this:

**C#**

```
[DebuggerDisplay("Count={Count}")]
[DebuggerTypeProxy(typeof(ArrayDebugView))]
[Serializable()]
public partial class ArrayEx<T>
```

#### Visual Basic

```
<DebuggerDisplay("Count={Count}")> _
<DebuggerTypeProxy(GetType(ArrayDebugView))> _
<Serializable()> _
Partial Public Class ArrayEx(Of T)
```

And then mark the *m\_syncRoot* field as nonserializable as follows.

**C#**

```
[NonSerialized()]
object m_syncRoot;
```

#### Visual Basic

```
<NonSerialized()> _
Private m_syncRoot As Object
```

Your *ArrayEx(T)* class can now be serialized. There is no need to serialize the synchronization object because it is only an object created with *new object()*. The *deserialize* method just calls *new object()* when it determines that the *m\_syncRoot* needs to be deserialized. So serializing *m\_syncRoot* makes the serialized data larger.

## The *ArrayEx(T)* Class

You might have noticed a problem in the previous section with serializing the *m\_data* field. The *Serializable* attribute serializes all elements in the *m\_data* field instead of the ones currently being used. For example, if you add 1,000 items and then remove them, the serializer still serializes 1,000 elements. You can avoid this by implementing the *ISerializable* interface and specifying the items to serialize instead. To do this, add a class named *ArrayEx.Serialize.cs* to your Microsoft Visual C# project or add *ArrayEx.Serialize.vb* to your Microsoft Visual Basic project, and change the following declaration from this:

**C#**

```
class ArrayEx
{}
```

#### Visual Basic

```
Public Class ArrayEx
End Class
```

to the following:

**C#**

```
[Serializable()]
public partial class ArrayEx<T> : ISerializable
{
}
```

**Visual Basic**

```
<Serializable()> _
Partial Public Class ArrayEx
    Implements ISerializable

End Class
```

Add the following constructor.

**C#**

```
private ArrayEx(SerializationInfo info, StreamingContext context)
{
    m_count = info.GetInt32("count");
    m_data = (T[])info.GetValue("data", typeof(T[]));
}
```

**Visual Basic**

```
Private Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
    m_count = info.GetInt32("count")
    m_data = CType(info.GetValue("data", GetType(T())), T())
End Sub
```

Add the method *GetObjectData*.

**C#**

```
void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.AddValue("count", m_count);

#if ALWAYS_SERIALIZE_INTERNAL_ARRAY
    info.AddValue("data", m_data);
#else
    // Always serialize multiples of GROW_BY
    int sizeToSerialize = (m_count % GROW_BY == 0) ? m_count : m_count +
        (GROW_BY - m_count % GROW_BY);

    if (m_count >= m_data.Length - GROW_BY)
    {
        info.AddValue("data", m_data);
    }
    else
```

```

    {
        T[] tmp = new T[sizeToSerialize];
        Array.Copy(m_data, tmp, sizeToSerialize);
        info.AddValue("data", tmp);
    }
#endif
}

```

### Visual Basic

```

Sub GetObjectData(ByVal info As SerializationInfo, ByVal context As StreamingContext)
Implements ISerializable.GetObjectData
    info.AddValue("count", m_count)

#If ALWAYS_SERIALIZE_INTERNAL_ARRAY Then
    info.AddValue("data", m_data)
#Else
    ' Always serialize multiples of GROW_BY
    Dim sizeToSerialize As Integer = IIf((m_count Mod GROW_BY) = 0, m_count, m_count + _
                                         (GROW_BY - m_count Mod GROW_BY))

    If (m_count >= m_data.Length - GROW_BY) Then
        info.AddValue("data", m_data)
    Else
        Dim tmp As T() = New T(sizeToSerialize - 1) {}
        Array.Copy(m_data, tmp, sizeToSerialize)
        info.AddValue("data", tmp)
    End If
#End If
End Sub

```

With *ALWAYS\_SERIALIZE\_INTERNAL\_ARRAY*, you can specify whether you want optimized serialization. A temporary variable has to be created to serialize the data. This can be a performance problem if you do a lot of serializing and/or you have very large arrays. Optimizing the serialization can be useful when serializing data types but not when serializing reference types, because items are nulled with reference types. Setting the *Capacity* to *Count* is another option if you need to serialize only every now and then.

## The Linked List Classes

You need to store each node and the node it points to in the linked list. The easiest way to do this is to save each node in the order each appears in the linked list. You then only need to add each node to the new linked list in the order you retrieve them. This causes each node to point to the correct node. You can obtain an array of node values by calling the *ToArrayList* method you created for the linked list class in Chapter 1, “Understanding Collections: Arrays and Linked Lists.”

Add a class named *SingleLinkedList.Serialize.cs* and *DoubleLinkedList.Serialize.cs* to your C# project and *SingleLinkedList.Serialize.vb* and *DoubleLinkedList.Serialize.vb* to your Visual Basic project.

Change the class definition in *SingleLinkedList.Serialize.cs* or *SingleLinkedList.Serialize.vb* from the following:

**C#**

```
class SingleLinkedList
{
}
```

**Visual Basic**

```
Public Class SingleLinkedList
```

```
End Class
```

to this:

**C#**

```
[Serializable]
public partial class SingleLinkedList<T> : ISerializable
{
    private SingleLinkedList(SerializationInfo info, StreamingContext context)
    {
        int count = info.GetInt32("count");

        if (count > 0)
        {
            T[] tmp = (T[])info.GetValue("nodes", typeof(T[]));
            for (int i = 0; i < count; ++i)
            {
                AddToEnd(tmp[i]);
            }
        }
    }

    #region ISerializable Members

    void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("count", Count);

        if (Count > 0)
        {
            T[] tmp = ToArray();

            info.AddValue("nodes", tmp);
        }
    }

    #endregion
}
```

**Visual Basic**

```
<Serializable()> _
Partial Public Class SingleLinkedList(Of T)
    Implements ISerializable

    Private Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
        Dim count As Integer = info.GetInt32("count")

        If (Count > 0) Then
            Dim tmp As T() = CType(info.GetValue("nodes", GetType(T())), T())
            For i As Integer = 0 To count - 1
                AddToEnd(tmp(i))
            Next
        End If
    End Sub

    Sub GetObjectData(ByVal info As SerializationInfo, ByVal context As StreamingContext) _
        Implements ISerializable.GetObjectData
        info.AddValue("count", Count)

        If (Count > 0) Then
            Dim tmp As T() = ToArray()

            info.AddValue("nodes", tmp)
        End If
    End Sub

End Class
```

Change the class definition in *DoubleLinkedList.Serialize.cs* or *DoubleLinkedList.Serialize.vb* from:

**C#**

```
class DoubleLinkedList
{ }
```

**Visual Basic**

```
Public Class DoubleLinkedList
```

```
End Class
```

to the following:

**C#**

```
[Serializable]
public partial class DoubleLinkedList<T>: ISerializable
{
    private DoubleLinkedList(SerializationInfo info, StreamingContext context)
    {
        int count = info.GetInt32("count");

        if (count > 0)
```

```

    {
        T[] tmp = (T[])info.GetValue("nodes", typeof(T[]));
        for (int i = 0; i < count; ++i)
        {
            AddToEnd(tmp[i]);
        }
    }

#region ISerializable Members

void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.AddValue("count", Count);

    if (Count > 0)
    {
        T[] tmp = ToArray();

        info.AddValue("nodes", tmp);
    }
}

#endregion
}

```

### Visual Basic

```

<Serializable()> _
Partial Public Class DoubleLinkedList(Of T)
    Implements ISerializable

    Private Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
        Dim count As Integer = info.GetInt32("count")

        If (Count > 0) Then
            Dim tmp As T() = CType(info.GetValue("nodes", GetType(T())), T())
            For i As Integer = 0 To count - 1
                AddToEnd(tmp(i))
            Next
        End If
    End Sub

    Sub GetObjectData(ByVal info As SerializationInfo, ByVal context As StreamingContext) _
        Implements ISerializable.GetObjectData
        info.AddValue("count", Count)

        If (Count > 0) Then
            Dim tmp As T() = ToArray()

            info.AddValue("nodes", tmp)
        End If
    End Sub

End Class

```

Then mark the *m\_syncRoot* field as nonserializable as follows.

**C#**

```
[NonSerialized()]
object m_syncRoot;
```

**Visual Basic**

```
<NonSerialized()> _
Private m_syncRoot As Object
```

## The Associative Array Classes

The following sections show you how to add serialization support to the two associative array classes you created in Chapter 2, "Understanding Collections: Associative Arrays."

### The *AssociativeArrayAL(TKey,TValue)* Class

Because *AssociativeArrayAL(TKey,TValue)* uses *DoubleLinkedList(TKey,TValue)* for internal storage, you need only to add *SerializableAttribute* to the *AssociativeArrayAL(TKey,TValue)* class and to the *AssociativeArrayAL(TKey,TValue).KVPair* structure.

Add a class named *AssociativeArrayAL.Serialize.cs* to your C# project and *AssociativeArrayAL.Serialize.vb* to your Visual Basic project. Then change the following:

**C#**

```
class AssociativeArrayAL
{
}
```

**Visual Basic**

```
Public Class AssociativeArrayAL
```

```
End Class
```

to this:

**C#**

```
[Serializable()]
public partial class AssociativeArrayAL<TKey, TValue>
{
}
```

**Visual Basic**

```
<Serializable()> _
Partial Public Class AssociativeArrayAL(Of TKey, TValue)
```

```
End Class
```

Also change the *KVPair* structure in *AssociativeArrayAL(TKey,TValue)* class from this:

**C#**

```
/// <summary>
/// Used for storing a key value pair.
/// </summary>
private struct KVPair
```

**Visual Basic**

```
'<summary>
'Used for storing a key value pair.
'</summary>
Private Structure KVPair
```

to the following:

**C#**

```
/// <summary>
/// Used for storing a key value pair.
/// </summary>
[Serializable()]
private struct KVPair
```

**Visual Basic**

```
'<summary>
'Used for storing a key value pair.
'</summary>
<Serializable()> _
Private Structure KVPair
```

Then mark the *m\_syncRoot* field as nonserializable as follows.

**C#**

```
[NonSerialized()]
object m_syncRoot;
```

**Visual Basic**

```
<NonSerialized()> _
Private m_syncRoot As Object
```

## The *AssociativeArrayHT* Class

Because the hash table bucket size and entries are both set to a prime number, using the default serializer could create a very large serialized object. You need to create a custom serializer to ensure that the minimum amount of data is serialized.

Add a class named *AssociativeArrayHT.Serialize.cs* to your C# project or *AssociativeArrayHT.Serialize.vb* to your Visual Basic project and then change the following:

**C#**

```
class AssociativeArrayHT
{
}
```

**Visual Basic**

```
Public Class AssociativeArrayHT
End Class
```

to this:

**C#**

```
[Serializable()]
public partial class AssociativeArrayHT<TKey, TValue> : ISerializable
{
    private AssociativeArrayHT(SerializationInfo info, StreamingContext context)
    {
        int count = info.GetInt32("count");

        if (count > 0)
        {
            List<KeyValuePair<TKey, TValue>> tmp =
                (List<KeyValuePair<TKey, TValue>>)info.GetValue("kvps",
                    typeof(List<KeyValuePair<TKey, TValue>>));
            foreach (KeyValuePair<TKey, TValue> kvp in tmp)
            {
                Add(kvp.Key, kvp.Value);
            }
        }
    }

    #region ISerializable Members

    void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("count", Count);

        if (Count > 0)
        {
            List<KeyValuePair<TKey, TValue>> kvps = new List<KeyValuePair<TKey, TValue>>();

            foreach (KeyValuePair<TKey, TValue> kvp in this)
            {
                kvps.Add(kvp);
            }

            info.AddValue("kvps", kvps);
        }
    }

    #endregion
}
```

**Visual Basic**

```
<Serializable()> _
Partial Public Class AssociativeArrayHT(Of TKey, TValue)
    Implements ISerializable

    Private Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
        Dim count As Integer = info.GetInt32("count")

        If (count > 0) Then
            Dim tmp As List(Of KeyValuePair(Of TKey, TValue)) = _
                CType(info.GetValue("kvps", _
                    GetType(List(Of KeyValuePair(Of TKey, TValue)))), _
                    List(Of KeyValuePair(Of TKey, TValue)))
            For Each kvp As KeyValuePair(Of TKey, TValue) In tmp
                Add(kvp.Key, kvp.Value)
            Next
        End If
    End Sub

    Sub GetObjectData(ByVal info As SerializationInfo, ByVal context As StreamingContext) _
        Implements ISerializable.GetObjectData
        info.AddValue("count", Count)

        If (Count > 0) Then
            Dim kvps As List(Of KeyValuePair(Of TKey, TValue)) = _
                New List(Of KeyValuePair(Of TKey, TValue))()
            For Each kvp As KeyValuePair(Of TKey, TValue) In Me
                kvps.Add(kvp)
            Next
            info.AddValue("kvps", kvps)
        End If
    End Sub
End Class
```

Then mark the *m\_syncRoot* field as nonserializable as follows.

**C#**

```
[NonSerialized()]
object m_syncRoot;
```

**Visual Basic**

```
<NonSerialized()> _
Private m_syncRoot As Object
```

The preceding code enumerates the *AssociativeArrayHT(TKey,TValue)* and stores each *KeyValuePair(TKey,TValue)* when it is time to serialize. The deserializer then retrieves each *KeyValuePair(TKey,TValue)* and adds them to *AssociativeArrayHT(TKey,TValue)* by calling the *Add* method.

## The Queue Classes

To add serialization support to the Queue classes, you need to add only the *Serializable* attribute to the class definition.

Add a class named *QueuedArray.Serialize.cs* and another class named *QueuedLinkedList.Serialize.cs* to your C# project, or *QueuedArray.Serialize.vb* and another named *QueuedLinkedList.Serialize.vb* to your Visual Basic project.

Change the class definition in *QueuedArray.Serialize.cs* or *QueuedArray.Serialize.vb* from the following:

**C#**

```
class QueuedArray
{
}
```

**Visual Basic**

```
Class QueuedArray
End Class
```

to this:

**C#**

```
[Serializable()]
public partial class QueuedArray<T>
{}
```

**Visual Basic**

```
<Serializable()> _
Partial Public Class QueuedArray(Of T)
End Class
```

Change the class definition in *QueuedLinkedList.Serialize.cs* or *QueuedLinkedList.Serialize.vb* from this:

**C#**

```
class QueuedLinkedList
{}
```

**Visual Basic**

```
Class QueuedLinkedList
End Class
```

to the following:

**C#**

```
[Serializable()]
public partial class QueuedLinkedList<T>
{
}
```

**Visual Basic**

```
<Serializable()> _
Partial Public Class QueuedLinkedList(Of T)

End Class
```

Then mark the *m\_syncRoot* field as nonserializable as follows.

**C#**

```
[NonSerialized()]
object m_syncRoot;
```

**Visual Basic**

```
<NonSerialized()> _
Private m_syncRoot As Object
```

## The Stack Classes

To add support for serialization to the Stack classes, you need to add only the *Serializable* attribute to the class definition.

Add a class named *StackedArray.Serialize.cs* and *StackedLinkedList.Serialize.cs* to your C# project or *StackedArray.Serialize.vb* and *StackedLinkedList.Serialize.vb* to your Visual Basic project.

Change the class definition in *StackedArray.Serialize.cs* or *StackedArray.Serialize.vb* from the following:

**C#**

```
class StackedArray
{
}
```

**Visual Basic**

```
Class StackedArray
```

```
End Class
```

to this:

**C#**

```
[Serializable()]
public partial class StackedArray<T>
{}
```

**Visual Basic**

```
<Serializable()> _
Partial Public Class StackedArray(Of T)

End Class
```

Change the class definition in *StackedLinkedList.Serialize.cs* or *StackedLinkedList.Serialize.vb* from the following:

**C#**

```
class StackedLinkedList
{}
```

**Visual Basic**

```
Class StackedLinkedList
```

```
End Class
```

to this:

**C#**

```
[Serializable()]
public partial class StackedLinkedList<T>
{}
```

**Visual Basic**

```
<Serializable()> _
Partial Public Class StackedLinkedList(Of T)
```

```
End Class
```

Then mark the *m\_syncRoot* field as nonserializable as follows.

**C#**

```
[NonSerialized()]
object m_syncRoot;
```

**Visual Basic**

```
<NonSerialized()> _
Private m_syncRoot As Object
```

## Summary

In this chapter, you saw how to add serialization support to the collection classes you created in Chapter 1 through Chapter 3, “Understanding Collections: Queues, Stacks, and Circular Buffers.” You also saw how to serialize the collection classes you worked with in Chapter 1 through Chapter 5, “Generic and Support Collections.” Using the information you learned in this chapter, you can serialize collections and also control the data being serialized.



Part IV

# Using Collections with UI Controls



## Chapter 10

# Using Collections with Windows Form Controls

After completing this chapter, you will be able to

- Perform simple bindings with the controls.
- Create two-way bindings with the controls.
- Understand the sample code.

## Simple Binding

Some Windows Form controls can bind to collections that implement *IList* and *IEnumerable* by setting the *DataSource* property of the *ListBox*, *ComboBox*, or *DataGridView* classes to the collection; the following shows an example.

C#

```
ListBox lb = new ListBox();  
  
lb.DataSource = new int [] { 1,2,3,4,5 };
```

Visual Basic

```
Dim lb As New ListBox()  
  
lb.DataSource = New Integer() {1, 2, 3, 4, 5}
```

To control which property of the items in the collection these controls should display, you set the *DisplayMember* property of the *ComboBox* or *ListBox* control, as follows.

C#

```
ComboBox cb = new ComboBox();  
  
cb.DataSource = new Company[]  
{  
    new Company()  
    {  
        Id = 1,  
        Name = "Alpine Ski House ",  
        Website = "http://www.alpineskihouse.com/"  
    },  
    new Company()
```

```

    {
        Id = 2,
        Name = "Tailspin Toys",
        Website = "http://www.tailspintoys.com/"
    }
};

cb.DisplayMember = "Name";

```

**Visual Basic**

```

Dim cb As New ComboBox()

cb.DataSource = New Company() _
{
    New Company() With _
    {
        .Id = 1, _
        .Name = "Alpine Ski House ", _
        .Website = "http://www.alpineskihouse.com/" _
    },
    New Company() With _
    {
        .Id = 2, _
        .Name = "Tailspin Toys", _
        .Website = "http://www.tailspintoys.com/" _
    }
}
cb.DisplayMember = "Name"

```

You can also control which property the controls use for the value of one or more selected items for *ComboBox* or *ListBox* controls by using the *ValueMember* property, as follows.

**C#**

```
cb.ValueMember = "Id";
```

**Visual Basic**

```
cb.ValueMember = "Id"
```

When a user selects an item or items, the control's *SelectedValue* property then contains the *Id* property of the selected item or items.

## Two-Way Data Binding

You may have noticed that if you update the collection that you assigned to the *DataSource* property of a control, the control doesn't reflect the change. That happens when the data source isn't a *BindingSource* or doesn't implement *IBindingList*. To support two-way binding, you must set *DataSource* to an object that implements *IBindingList* or use the *BindingSource*.



**Note** You must perform all *Remove*, *Clear*, and *Add* method calls through *BindingSource* if you specify a data source that doesn't implement *IBindingList*. Otherwise, the bound item will not see your changes.

## Implementing the *IBindingList* Interface

*IBindingList* provides both simple and complex support for binding to data sources. The interface allows you to notify bound items of list changes and provides support for simple searching and sorting of the underlying collection. You should look at *IBindingListView* if you need to do complex sorting or add filtering. Microsoft provides *BindingList(T)* in the Microsoft .NET Framework so that you do not have to implement an *IBindingList* from scratch.



**Note** The sample code in the DevGuideToCollections project, in either the Chapter 10\CS\DevGuideToCollections folder or Chapter 10\VB\DevGuideToCollections folder, contains a full implementation of the *IBindingList* interface called *WinFormsBindingList(T)*. The implementation of each interface implemented by *WinFormsBindingList(T)* is broken out into separate files to make it easier to follow. You'll find the implementation of *IBindingList* in the language-specific files *WinFormsBindingList.BindingList.cs* and *WinFormsBindingList.BindingList.vb*. You can review those files for the full implementation of the properties and methods discussed later in this section. Chapter 6, ".NET Collection Interfaces," and Chapter 8, "Using Threads with Collections," provide information about how to implement the other interfaces. *WinFormsBindingList(T)*, which you wrote in Chapter 1, "Understanding Collections: Arrays and Linked Lists," is the *ArrayEx(T)* class modified to support the *IBindingList* interface.

## Adding List Manipulation Support

Bound items can determine whether a collection can be modified by using the *AllowEdit*, *AllowNew*, and *AllowRemove* properties, and by using the *AddNew* method.

**AllowEdit** This property returns a value that states whether items in the list can be updated. The *WinFormsBindingList(T)* returns *true* if the items contained in the collection support *INotifyPropertyChanged*. The *DataGridView* doesn't allow editing of the items if the *AllowEdit* property is *false*. So the *WinFormsBindingList* will not allow users to modify the items if the items do not provide property change notifications.

**AllowNew** This property returns a value that states whether items can be added through the *AddNew* method. The property should return *false* if the *IList.IsFixedSize* or *IList.IsReadOnly* property is set to *true*.

**AllowRemove** This property returns a value that states whether items can be removed from the collection. The property should return *false* if the *IList.IsFixedSize* or *IList.IsReadOnly* property is set to *true*.

**AddNew** This method adds a new item to the list. It throws an exception if the *AllowNew* property is false. The following code shows the implementation contained in *WinFormsBindingList(T)*.

#### C#

```
public object AddNew()
{
    if (!AllowNew)
    {
        throw new InvalidOperationException();
    }

    T retval = Activator.CreateInstance<T>();

    Add(retval);

    m_newIndex = Count - 1;

    return retval;
}
```

#### Visual Basic

```
Public Function AddNew() As Object Implements IBindingList.AddNew
    If (Not AllowNew) Then
        Throw New InvalidOperationException()
    End If

    Dim retval As T = Activator.CreateInstance(Of T)()

    Add(retval)

    m_newIndex = Count - 1

    Return retval
End Function
```

The field *m\_newIndex* holds the index of the item being added. This is needed so that the bound item, such as the *DataGridView*, can cancel a newly added item. This is accomplished through the *ICancelAddNew* interface. The *ICancelAddNew* interface implementation used in *WinFormsBinding(T)* is as follows.

#### C#

```
public void CancelNew(int itemIndex)
{
    if (m_newIndex.HasValue && m_newIndex.Value == itemIndex)
    {
        RemoveAt(itemIndex);
    }
}
```

```
public void EndNew(int itemIndex)
{
    if (m_newIndex.HasValue && m_newIndex.Value == itemIndex)
    {
        m_newIndex = null;
    }
}
```

### Visual Basic

```
Public Sub CancelNew(ByVal itemIndex As Integer) Implements ICancelAddNew.CancelNew
    If (m_newIndex.HasValue And m_newIndex.Value = itemIndex) Then
        RemoveAt(itemIndex)
    End If
End Sub

Public Sub EndNew(ByVal itemIndex As Integer) Implements ICancelAddNew.EndNew
    If (m_newIndex.HasValue And m_newIndex.Value = itemIndex) Then
        m_newIndex = Nothing
    End If
End Sub
```

If a user starts a new row in the *DataGridView* and then clicks elsewhere, the *DataGridView* calls *ICancelAddNew.CancelNew*; otherwise it calls *ICancelAddNew.EndNew* after the user enters data in the new row. The *CancelNew* implementation removes the newly added item from the collection. The *EndNew* implementation sets *m\_newIndex* to *null* to denote that the item has been committed.

## Adding Notification Support

*IBindingList* provides list change notifications to items bound to the collection. Bound items can check to see whether the collection sends notifications through the *SupportsChangeNotification* property, and they can receive those notifications through the *ListChanged* event. Bound items can then inspect the *ListChangedEventArgs* to see what has changed in the list. Bound items are also notified of property changes to the items in the collection through the *ListChanged* event.

The collection must raise the *ListChanged* event whenever the list is modified. This occurs whenever the *Add*, *Clear*, *Insert*, *Remove*, or *RemoveAt* methods are called as well as when the *Item* set property is called. The easiest way to do this is by creating *OnXXX* and *OnXXXComplete* methods like those discussed in the "Using *CollectionBase*" section in Chapter 5, "Generic and Support Collections." Each method except the *Add* method has *OnXXX* and *OnXXXComplete* methods. The *Add* and *Insert* methods use the *OnInsert* and *OnInsertComplete* methods; the *RemoveAt* and *Remove* methods use the *OnRemove* and *OnRemoveComplete* methods. The *OnXXX* method is called at the beginning of the operation, and the *OnXXXComplete* is called at the end of the method. The following code shows how the *Add*, *Clear*, *Remove*, *RemoveAt*, and *Insert* methods as well as the *Item* property are copied from *ArrayEx(T)* and modified to support the *OnXXX* and *OnXXXComplete* methods.

**C#**

```
public void Add(T item)
{
    OnInsert(item, Count);
    InnerList.Add(item);
    OnInsertComplete(item, Count - 1);
}
public void Clear()
{
    var removed = ToArray();
    OnClear(removed);
    InnerList.Clear();
    OnClearComplete(removed);
}
public bool Remove(T item)
{
    if (!AllowRemove)
    {
        throw new NotSupportedException();
    }

    int index = IndexOf(item);

    if (index >= 0)
    {
        OnRemove(item, index);

        InnerList.Remove(item);

        OnRemoveComplete(item, index);
    }

    return index >= 0;
}
public void RemoveAt(int index)
{
    if (index < 0 || index >= Count)
    {
        // Item has already been removed.
        return;
    }

    if (!AllowRemove)
    {
        throw new NotSupportedException();
    }

    if (IsSorted)
```

```
{           throw new NotSupportedException("You cannot remove by index on a sorted list.");
}

T item = m_data[index];

OnRemove(item, index);

InnerList.RemoveAt(index);

OnRemoveComplete(item, index);
}
public void Insert(int index, T item)
{
    OnInsert(item, index);

    InnerList.Insert(index, item);

    OnInsertComplete(item, index);
}
public T this[int index]
{
    set
    {
        T oldValue = InnerList[index];

        OnSet(index, oldValue, value);

        InnerList[index] = value;

        OnSetComplete(index, oldValue, value);
    }
}
```

### Visual Basic

```
Public Sub Add(ByVal item As T) Implements ICollection(Of T).Add
    OnInsert(item, Count)
    InnerList.Add(item)
    OnInsertComplete(item, Count - 1)
End Sub

Public Sub Clear() Implements ICollection(Of T).Clear, IList.Clear
    Dim removed As T() = ToArray()
    OnClear(removed)
    InnerList.Clear()
    OnClearComplete(removed)
End Sub

Public Function Remove(ByVal item As T) As Boolean Implements ICollection(Of T).Remove
    If (Not AllowRemove) Then
        Throw New NotSupportedException()
    End If
```

```
Dim index As Integer = IndexOf(item)

If (index >= 0) Then
    OnRemove(item, index)

    InnerList.Remove(item)

    OnRemoveComplete(item, index)
End If

Return index >= 0
End Function

Public Sub RemoveAt(ByVal index As Integer) Implements IList(Of T).RemoveAt, IList.RemoveAt
If (index < 0 Or index >= Count) Then
    ' Item has already been removed.
    Return
End If

If (Not AllowRemove) Then
    Throw New NotSupportedException()
End If

If (IsSorted) Then
    Throw New NotSupportedException("You cannot remove by index on a sorted list.")
End If

Dim item As T = InnerList(index)

OnRemove(item, index)

InnerList.RemoveAt(index)

OnRemoveComplete(item, index)
End Sub

Public Sub Insert(ByVal index As Integer, ByVal item As T) Implements IList(Of T).Insert
OnInsert(item, index)

InnerList.Insert(index, item)

OnInsertComplete(item, index)
End Sub

Default Public Property Item(ByVal index As Integer) As T Implements IList(Of T).Item
Set(ByVal value As T)
    Dim oldValue As T = InnerList(index)

    OnSet(index, oldValue, value)

    InnerList(index) = value

    OnSetComplete(index, oldValue, value)
End Set
End Property
```

The *OnXXX* methods check whether the corresponding method can be called, such as when a user calls *Insert* instead of *Add* on a sorted list.

### C#

```
void OnSet(int index, T oldValue, T newValue)
{
    if (IsSorted)
    {
        throw new NotSupportedException("You cannot set items in a sorted list");
    }
}
void OnClear(T[] itemsRemoved)
{
}
void OnInsert(T item, int index)
{
    // You can only add to the end of the list when sorting is on
    if (IsSorted && index != Count)
    {
        throw new NotSupportedException();
    }
}
void OnRemove(T item, int index)
{
}
```

### Visual Basic

```
Sub OnSet(ByVal index As Integer, ByVal oldValue As T, ByVal newValue As T)
    If (IsSorted) Then
        Throw New NotSupportedException("You cannot set items in a sorted list")
    End If
End Sub

Sub OnClear(ByVal itemsRemoved As T())
End Sub

Sub OnInsert(ByVal item As T, ByVal index As Integer)
    ' You can only add to the end of the list is sorting is on
    If (IsSorted And index <> Count) Then
        Throw New NotSupportedException()
    End If
End Sub

Sub OnRemove(ByVal item As T, ByVal index As Integer)
End Sub
```

To keep it simple, the *WinFormsBindingList(T)* class throws an exception when a user tries to insert an item in a sorted list instead of trying to figure out whether the user should insert the item in the sorted or unsorted list. If you decide to implement insertion into a sorted list in your version, you should determine whether the user wants to insert into the sorted list she currently sees or the stored, unsorted list she sees when she makes a call such as *Insert(2,value)*. For example, if the unsorted list is [3,1,2,7,6,5] and the sorted list the user sees

is  $[1,2,3,5,6,7]$ , where should 4 be inserted in the unsorted list if the user calls *Insert(3,4)*? Does the user want you to insert 4 into the unsorted list at index 3 because the unsorted list is completely different?

The *OnXXXComplete* methods are responsible for notifying the bound item of list changes, registering for property changes, and handling special cases after an add or removal, such as re-indexing or re-sorting the list.

C#

```
void OnSetComplete(int index, T oldValue, T newValue)
{
    UnRegisterForPropertyChanges(oldValue);
    RegisterForPropertyChanges(newValue);

    if (SupportsSearching)
    {
        ReIndex();
    }

    OnListChanged(new ListChangedEventArgs(ListChangedType.ItemChanged, index));
}

void OnClearComplete(T[] itemsRemoved)
{
    foreach (var item in itemsRemoved)
    {
        UnRegisterForPropertyChanges(item);
    }

    if (SupportsSearching)
    {
        ReIndex();
    }

    if (_originalList != null)
    {
        _originalList.Clear();
    }

    OnListChanged(new ListChangedEventArgs(ListChangedType.Reset, -1));
}

void OnInsertComplete(T item, int index)
{
    RegisterForPropertyChanges(item);
    if (IsSorted)
    {
        Sort();
    }
    else
```

```
{  
    if (SupportsSearching)  
    {  
        ReIndex();  
    }  
  
    OnListChanged(new ListChangedEventArgs(ListChangedType.ItemAdded, index));  
}  
  
if (_originalList != null)  
{  
    _originalList.Insert(index, item);  
}  
}  
}  
void OnRemoveComplete(T item, int index)  
{  
    UnRegisterForPropertyChanges(item);  
    if (SupportsSearching)  
    {  
        ReIndex();  
    }  
    OnListChanged(new ListChangedEventArgs(ListChangedType.ItemDeleted, index));  
  
    if (_originalList != null)  
    {  
        _originalList.Remove(item);  
    }  
}
```

### Visual Basic

```
Sub OnSetComplete(ByVal index As Integer, ByVal oldValue As T, ByVal newValue As T)  
  
    UnRegisterForPropertyChanges(oldValue)  
    RegisterForPropertyChanges(newValue)  
  
    If (SupportsSearching) Then  
        ReIndex()  
    End If  
  
    OnListChanged(New ListChangedEventArgs(ListChangedType.ItemChanged, index))  
End Sub  
  
Sub OnClearComplete(ByVal itemsRemoved As T())  
    For Each item As T In itemsRemoved  
        UnRegisterForPropertyChanges(item)  
    Next  
  
    If (SupportsSearching) Then  
        ReIndex()  
    End If
```

```

If (_originalList IsNot Nothing) Then
    _originalList.Clear()
End If
OnListChanged(New ListChangedEventArgs(ListChangedType.Reset, -1))
End Sub

Sub OnInsertComplete(ByVal item As T, ByVal index As Integer)
    RegisterForPropertyChanges(item)
    If (IsSorted) Then
        Sort()
    Else
        If (SupportsSearching) Then
            ReIndex()
        End If
    End If

    OnListChanged(New ListChangedEventArgs(ListChangedType.ItemAdded, index))
End If

If (_originalList IsNot Nothing) Then
    _originalList.Insert(index, item)
End If
End Sub

Sub OnRemoveComplete(ByVal item As T, ByVal index As Integer)
    UnRegisterForPropertyChanges(item)
    If (SupportsSearching) Then
        ReIndex()
    End If

    OnListChanged(New ListChangedEventArgs(ListChangedType.ItemDeleted, index))

    If (_originalList IsNot Nothing) Then
        _originalList.Remove(item)
    End If
End Sub

```

*WinFormsBindingList(T)* gets property change notifications by registering to the *INotifyPropertyChanged* and *INotifyPropertyChanging* interfaces, as follows.

#### C#

```

void UnRegisterForPropertyChanges(T item)
{
    // No need to register for property changes if none of the
    // IBindinglist features are supported
    if ((!_supportsPropertyChanged && !_supportsPropertyChanging) ||
        (!SupportsSorting && !SupportsChangeNotification && !SupportsSearching))
    {
        return;
    }

    INotifyPropertyChanged changed = item as INotifyPropertyChanged;
    INotifyPropertyChanging changing = item as INotifyPropertyChanging;

    if (changing != null && SupportsSearching)

```

```

{
    changing.PropertyChanging -= new PropertyChangingEventHandler(T_OnPropertyChanging);
}
if (changed != null)
{
    changed.PropertyChanged -= new PropertyChangedEventHandler(T_OnPropertyChanged);
}
}

void RegisterForPropertyChanges(T item)
{
    // No need to register for property changes if none of the
    // IBindingList features are supported
    if ((!m_supportsPropertyChanged && !m_supportsPropertyChanging) ||
        (!SupportsSorting && !SupportsChangeNotification && !SupportsSearching))
    {
        return;
    }

    INotifyPropertyChanged changed = item as INotifyPropertyChanged;
    INotifyPropertyChanging changing = item as INotifyPropertyChanging;

    if (changing != null && SupportsSearching)
    {
        changing.PropertyChanging += new PropertyChangingEventHandler(T_OnPropertyChanging);
    }
    if (changed != null)
    {
        changed.PropertyChanged += new PropertyChangedEventHandler(T_OnPropertyChanged);
    }
}

```

### Visual Basic

```

Sub UnRegisterForPropertyChanges(ByVal item As T)
    ' No need to register for property changes if none of the
    ' IBindingList features are supported
    If ((Not m_supportsPropertyChanged And Not m_supportsPropertyChanging) Or _
        (Not SupportsSorting And Not SupportsChangeNotification And Not SupportsSearching)) _
        Then
            Return
    End If

    Dim changed As INotifyPropertyChanged = TryCast(item, INotifyPropertyChanged)
    Dim changing As INotifyPropertyChanging = TryCast(item, INotifyPropertyChanging)

    If (changing IsNot Nothing And SupportsSearching) Then
        RemoveHandler changing.PropertyChanging, _
            New PropertyChangingEventHandler(AddressOf T_OnPropertyChanging)
    End If
    If (changed IsNot Nothing) Then
        RemoveHandler changed.PropertyChanged, _
            New PropertyChangedEventHandler(AddressOf T_OnPropertyChanged)
    End If
End Sub

```

```

Sub RegisterForPropertyChanges(ByVal item As T)
    ' No need to register for property changes if none of the
    ' IBindingList features are supported
    If ((Not m_supportsPropertyChanged And Not m_supportsPropertyChanging) Or _
        (Not SupportsSorting And Not SupportsChangeNotification And Not SupportsSearching)) _
    Then
        Return
    End If

    Dim changed As INotifyPropertyChanged = TryCast(item, INotifyPropertyChanged)
    Dim changing As INotifyPropertyChanging = TryCast(item, INotifyPropertyChanging)

    If (changing IsNot Nothing And SupportsSearching) Then
        AddHandler changing.PropertyChanging, _
                    New PropertyChangedEventHandler(AddressOf T_OnPropertyChanging)
    End If
    If (changed IsNot Nothing) Then
        AddHandler changed.PropertyChanged, _
                    New PropertyChangedEventHandler(AddressOf T_OnPropertyChanged)
    End If
End Sub

```

Property change notification is required internally for re-indexing and re-sorting the list if the property that changed is being indexed or sorted. Property change notifications are also needed so that bound controls can receive the *ListChangeType.ItemChange* notification.

The collection raises the *ListChanged* event through the *OnListChanged* method.

#### C#

```

void OnListChanged(ListChangedEventArgs e)
{
    if (!SupportsChangeNotification)
    {
        return;
    }

    if (ListChanged != null)
    {
        ListChanged(this, e);
    }
}

```

#### Visual Basic

```

Sub OnListChanged(ByVal e As ListChangedEventArgs)
    If (Not SupportsChangeNotification) Then
        Return
    End If

    RaiseEvent ListChanged(Me, e)
End Sub

```

Notification can be turned off, so the code checks the *SupportsChangeNotification* flag before raising the *ListChanged* event.

## Adding Sorting Support

*IBindingList* supports sorting through the *SupportsSorting*, *SortDirection*, *SortProperty*, and *IsSorted* properties as well as the *ApplySort* and *RemoveSort* methods. Internally, *IBindingList* maintains the original list so that it can be restored when sorting is removed. To do this, the *Add*, *Remove*, and *Clear* methods must add to both lists until sorting is removed.

**SupportsSorting** This property tells the bound item whether sorting is implemented.

**SortDirection** This property holds the current sort direction of the sort on the *SortProperty*. The property is not valid if *IsSorted* is *false*.

**SortProperty** This property states the currently sorted property. The property is not valid if *IsSorted* is *false*.

**IsSorted** This property states whether the contents are currently sorted. This will be *true* if *ApplySort* has been called and *RemoveSort* has not been called after the last *ApplySort*. The contents of *SortDirection* and *SortProperty* are valid only when *IsSorted* is *true*.

**ApplySort** This method applies the specified sort to the list and is implemented as follows.

C#

```
public void ApplySort(PropertyDescriptor property, ListSortDirection direction)
{
    if (!SupportsSorting)
    {
        throw new NotSupportedException();
    }

    if (!IsSorted)
    {
        if (m_originalList == null)
        {
            m_originalList = m_data;
            m_data = new List<T>(m_originalList);
        }
    }
    m_sortDirection = direction;
    m_sortDescriptor = property;
    IsSorted = true;
    Sort();
}
```

Visual Basic

```
Public Sub ApplySort( _
    ByVal [property] As PropertyDescriptor, _
    ByVal direction As ListSortDirection _
) Implements IBindingList.ApplySort
If (Not SupportsSorting) Then
    Throw New NotSupportedException()
End If
```

```

If (Not IsSorted) Then
    If (m_originalList Is Nothing) Then
        m_originalList = m_data
        m_data = New List(Of T)(m_originalList)
    End If
End If
m_sortDirection = direction
m_sortDescriptor = [property]
m_isSorted = True
Sort()
End Sub

```

Before a sort operation, the original list is saved and copied over to *m\_data*. The field *m\_data* is then sorted using the *Sort* method.

### C#

```

void Sort()
{
    if (m_sortDescriptor == null)
    {
        return;
    }

    m_data.Sort(
        new LambdaComparer<T>
        (
            (x, y) =>
            {
                object xValue = m_sortDescriptor.GetValue(x);
                object yValue = m_sortDescriptor.GetValue(y);

                if (m_sortDirection == ListSortDirection.Descending)
                {
                    return System.Collections.Comparer.Default.Compare(
                        xValue, yValue) * -1;
                }

                return System.Collections.Comparer.Default.Compare(xValue, yValue);
            }
        ));
}

if (SupportsSearching)
{
    ReIndex();
}

OnListChanged(new ListChangedEventArgs(ListChangedType.Reset, -1));
}

```

## Visual Basic

```
Function Compare(ByVal x As T, ByVal y As T) As Integer
    Dim xValue As Object = m_sortDescriptor.GetValue(x)
    Dim yValue = m_sortDescriptor.GetValue(y)
    If (m_sortDirection = ListSortDirection.Descending) Then
        Return System.Collections.Comparer.Default.Compare(xValue, yValue) * -1
    End If
    Return System.Collections.Comparer.Default.Compare(xValue, yValue)
End Function

Sub Sort()
    If (m_sortDescriptor Is Nothing) Then
        Return
    End If

    m_data.Sort(AddressOf Compare)

    If (SupportsSearching) Then
        ReIndex()
    End If

    OnListChanged(New ListChangedEventArgs(ListChangedType.Reset, -1))
End Sub
```

In the preceding code, the *Sort* method uses a lambda expression in the Microsoft Visual C# code and a *Compare* function in the Microsoft Visual Basic code to do the sorting. The lambda expression sort creates the following custom class that implements *IComparer(T)* and passes the *Compare* method call to the lambda expression.

## C#

```
class LambdaComparer<T> : IComparer<T>
{
    Func<T, T, int> m_compare;

    public LambdaComparer(Func<T, T, int> compareFunction)
    {
        if (compareFunction == null)
        {
            throw new ArgumentNullException("compareFunction");
        }

        m_compare = compareFunction;
    }

    public int Compare(T x, T y)
    {
        return m_compare(x,y);
    }
}
```

The *Sort* method then notifies the bound item of the collection change by raising the *ListChanged* event.

**RemoveSort** This method removes any applied sorting.

#### C#

```
public void RemoveSort()
{
    if (!SupportsSorting)
    {
        throw new NotSupportedException();
    }
    m_sortDescriptor = null;
    IsSorted = false;
    if (m_originalList != null)
    {
        m_data = m_originalList;
        m_originalList = null;
    }
    if (SupportsSearching)
    {
        ReIndex();
    }
    OnListChanged(new ListChangedEventArgs(ListChangedType.Reset, -1));
}
```

#### Visual Basic

```
Public Sub RemoveSort() Implements IBindingList.RemoveSort
    If (Not SupportsSorting) Then
        Throw New NotSupportedException()
    End If
    m_sortDescriptor = Nothing
    m_isSorted = False
    If (m_originalList IsNot Nothing) Then
        m_data = m_originalList
        m_originalList = Nothing
    End If
    If (SupportsSearching) Then
        ReIndex()
    End If
    OnListChanged(New ListChangedEventArgs(ListChangedType.Reset, -1))
End Sub
```

The original list is restored and then indexed. The bound item is then notified of the change.

## Adding Searching Support

Searching is supported through the *SupportsSearching* property and the *Find*, *AddIndex*, and *RemoveIndex* methods. The *WinFormsBindingList(T)* shows how to get started implementing indexing if your application needs it, but the vast majority of applications do not need to support indexing.

***SupportsSearching*** Bound items can call this property to determine whether a collection supports searching. The *SupportsSearching* property lets you know whether the collection implements the *Find* method—but not necessarily whether the collection supports indexing. Most collections do not support indexing, but the sample code for this book provides an example indexing implementation to get you started.

***Find*** The *Find* method searches for the item that contains the specified property that matches the specified key.

### C#

```
public int Find(PropertyDescriptor property, object key)
{
    if (!SupportsSearching)
    {
        throw new NotSupportedException();
    }

    IndexData? data = FindIndexData(property);

    // See if the property has been indexed
    if (data.HasValue)
    {
        if (data.Value.Indexes.ContainsKey(key))
        {
            var indexes = data.Value.Indexes[key];
            if (indexes.Count > 0)
            {
                return indexes[0];
            }
        }
    }

    return -1;
}

// Find the key by iterating over every element
for (int i = 0; i < Count; ++i)
{
    T item = m_data[i];

    try
```

```

    {
        object value = property.GetValue(item);

        if (System.Collections.Comparer.Default.Compare(value, key) == 0)
        {
            return i;
        }
    }
    catch
    {
    }
}

return -1;
}

```

**Visual Basic**

```

Public Function Find( _
    ByVal [property] As PropertyDescriptor, _
    ByVal key As Object _
) As Integer Implements IBindingList.Find
If (Not SupportsSearching) Then
    Throw New NotSupportedException()
End If

Dim data As Nullable(Of IndexData) = FindIndexData([property])

' See if the property has been indexed
If (data.HasValue) Then
    If (data.Value.Indexes.ContainsKey(key)) Then
        Dim indexes = data.Value.Indexes(key)
        If (indexes.Count > 0) Then
            Return indexes(0)
        End If
    End If

    Return -1
End If

' Find the key by iterating over every element
For i As Integer = 0 To Count - 1
    Dim item As T = m_data(i)

    Try
        Dim value = [property].GetValue(item)

        If (System.Collections.Comparer.Default.Compare(value, key) = 0) Then
            Return i
        End If
    Catch
    End Try
Next

Return -1
End Function

```

The *Find* method first checks to see whether the property has been indexed. If it has, it then uses the index table to find the item; otherwise, it performs a linear search for the item.

**AddIndex** This method adds a *PropertyDescriptor* to the list to be indexed. The implementation in *WinFormsBindingList(T)* is as follows.

C#

```
public void AddIndex(PropertyDescriptor property)
{
    IndexData ?data = FindIndexData(property);

    if (!data.HasValue)
    {
        if (_indexes == null)
        {
            _indexes = new List<WinFormsBindingList<T>.IndexData>();
        }

        _indexes.Add
        (
            new IndexData()
            {
                Indexes = new Dictionary<object, List<int>>(),
                PropertyDescriptor = property
            }
        );
    }

    ReIndex();
}

IndexData ?FindIndexData(PropertyDescriptor property)
{
    if (_indexes == null)
    {
        return null;
    }

    foreach (var data in _indexes)
    {
        if (data.PropertyDescriptor == property)
        {
            return data;
        }
    }

    return null;
}
```

### Visual Basic

```

Public Sub AddIndex(ByVal [property] As PropertyDescriptor) Implements IBindingList.AddIndex
    Dim data As Nullable(Of IndexData) = FindIndexData([property])

    If (Not data.HasValue) Then
        If (m_indexes Is Nothing) Then
            m_indexes = New List(Of WinFormsBindingList(Of T).IndexData)()
        End If
        m_indexes.Add( _
            New IndexData() With { _
                .Indexes = New Dictionary(Of Object, List(Of Integer))(), _
                .PropertyDescriptor = [property] _
            })
    End If
    ReIndex()
End Sub

Function FindIndexData(ByVal prop As PropertyDescriptor) As Nullable(Of IndexData)
    If (m_indexes Is Nothing) Then
        Return Nothing
    End If

    For Each data As IndexData In m_indexes
        If (data.PropertyDescriptor.Name = prop.Name) Then
            Return data
        End If
    Next

    Return Nothing
End Function

```

The *FindIndexData* method tries to locate the index of the collection item that belongs to the specified property. If the property isn't already indexed, it creates a new index, and then calls the *ReIndex* method to force a re-indexing of all of the data, as follows.

### C#

```

void ReIndex()
{
    if (m_indexes == null)
    {
        return;
    }

    // Remove the old indexes
    foreach (var index in m_indexes)
    {
        index.Indexes.Clear();
    }

    if (m_indexes.Count == 0 || !SupportsSearching || !m_canIndex)
    {
        return;
    }
}

```

```
// Iterate over each item and add the index to the collection
for (int i = 0; i < Count; ++i)
{
    T item = m_data[i];

    foreach (var data in m_indexes)
    {
        try
        {
            object value = data.PropertyDescriptor.GetValue(item);

            AddIndexData(data, value, i);
        }
        catch
        {
        }
    }
}
```

### Visual Basic

```
Sub ReIndex()
    If (m_indexes Is Nothing) Then
        Return
    End If

    ' Remove the old indexes
    For Each index As IndexData In m_indexes
        index.Indexes.Clear()
    Next

    If (m_indexes.Count = 0 Or Not SupportsSearching Or Not m_canIndex) Then
        Return
    End If

    ' Iterate over each item and add the index to the collection
    For i As Integer = 0 To Count - 1
        Dim item As T = InnerList(i)

        For Each Data As IndexData In m_indexes
            Try
                Dim value = Data.PropertyDescriptor.GetValue(item)

                AddIndexData(Data, value, i)
            Catch
            End Try
        Next
    Next
End Sub
```

*ReIndex* works by erasing all of the old indexed data and then traversing each item in the collection and indexing that item. As you can tell, this is not an efficient way of indexing the system, but it shows the basic idea behind the *AddIndex* and *RemoveIndex* methods.

**RemoveIndex** This method removes a *PropertyDescriptor* from the list of indexed properties. The implementation in *WinFormsBindingList(T)* is as follows.

#### C#

```
public void RemoveIndex(PropertyDescriptor property)
{
    if (m_indexes == null)
    {
        return;
    }

    IndexData? data = FindIndexData(property);

    if (data.HasValue)
    {
        m_indexes.Remove(data.Value);
    }

    ReIndex();
}
```

#### Visual Basic

```
Public Sub RemoveIndex(ByVal [property] As PropertyDescriptor) _
    Implements IBindingList.RemoveIndex
    If (m_indexes Is Nothing) Then
        Return
    End If

    Dim data As Nullable(Of IndexData) = FindIndexData([property])

    If (data.HasValue) Then
        m_indexes.Remove(data.Value)
    End If

    ReIndex()
End Sub
```

## Implementing the *IBindingListView* Interface

The *IBindingListView* interface extends the *IBindingList* interface by providing advanced sorting and filtering capabilities.



**Note** The sample code in the DevGuideToCollections project, in either the Chapter 10\CS\DevGuideToCollections folder or Chapter 10\VB\DevGuideToCollections folder, contains a full implementation of the *IBindingListView* interface called *WinFormsBindingListView(T)*. The implementation of each interface implemented by *WinFormsBindingListView(T)* is broken out into separate files to make the logic easier to follow. You'll find the implementation of *IBindingListView* in the language-specific files *WinFormsBindingListView.BindingListView.cs* and *WinFormsBindingListView.BindingListView.vb*. You can review those files for the full implementation of the properties and methods discussed later in this section. Chapters 6 and 8 describe how to implement the other interfaces. *WinFormsBindingListView(T)* is the *WinFormsBindingList(T)* class modified to support the *IBindingListView* interface. See the "Implementing the *IBindingList* Interface" section earlier in this chapter for information about implementing *IBindingList*.

## Adding Filtering Support

Filtering is accomplished by using the *SupportsFiltering* and *Filter* properties as well as the *RemoveFilter* method.

**SupportsFiltering** This property states whether the collection supports filtering.

**Filter** Set this property to filter a collection. The filter format for *WinFormsBindingListView(T)* is a small subset of the expression that you can pass to *DataColumn.Expression*. The allowed syntax is as follows.

```
Filter = [NOT] (PropertyName|(PropertyName]) (>|<|<>|<=|>=|=) (Value|'Value')  
[(AND|OR) Filter]
```

Using that syntax, you can write code such as the following

```
[Name] = 'Value'  
  
Name <> Value  
  
Name <= Value AND NOT IsDeleted  
  
NOT IsDeleted  
  
Name = 'Value' AND IsDeleted = False  
  
Name = 'Value' OR Count = 2  
  
Name = 'Value' OR Count = 2 OR Height = 36
```

The following code shows the implementation in *WinFormsBindingListView(T)*. Because this book is a guide for collections rather than parsing, you won't be learning the implementation in *FilterParser*. You can investigate lexer/parsers if you want to implement your own parsing.

**C#**

```

public string Filter
{
    get
    {
        return m_filter;
    }
    set
    {
        if (m_filter == value)
        {
            return;
        }

        m_filterRoot = FilterParser.Parse(value);
        m_filter = value;

        if (!string.IsNullOrEmpty(value))
        {
            // Something needs to be filtered

            if (!m_isFiltering)
            {
                ApplyFilter();
            }
            else
            {
                ReapplyFilter();
            }
        }
        else
        {
            RemoveFilterInternal();
        }
    }
}

```

**Visual Basic**

```

Public Property Filter() As String Implements IBindingListView.Filter
    Get
        Return m_filter
    End Get
    Set(ByVal value As String)
        If (m_filter = value) Then
            Return
        End If

        m_filterRoot = FilterParser.Parse(value)
        m_filter = value

        If (Not String.IsNullOrEmpty(value)) Then
            ' Something needs to be filtered

            If (Not m_isFiltering) Then
                ApplyFilter()
            End If
        End If
    End Set
End Property

```

```
    Else
        ReapplyFilter()
    End If
Else
    RemoveFilterInternal()
End If
End Set
End Property
```

The *Filter* property parses the specified string and then determines whether it should apply the filter for the first time, reapply it, or remove the current filter. If the filter is null or empty, the code calls the *RemoveFilterInternal* method to remove the filter.

### C#

```
void RemoveFilterInternal()
{
    if (!m_isFiltering)
    {
        return;
    }

    if (IsSorted)
    {
        m_data = new List<T>(m_originalList);
        Sort();
    }
    else
    {
        m_data = m_originalList;
        m_originalList = null;
        OnListChanged(new ListChangedEventArgs(ListChangedType.Reset, -1));
    }

    m_isFiltering = false;
}
```

### Visual Basic

```
Sub RemoveFilterInternal()
    If (Not m_isFiltering) Then
        Return
    End If

    If (IsSorted) Then
        m_data = New List(Of T)(m_originalList)
        Sort()
    Else
        m_data = m_originalList
        m_originalList = Nothing
        OnListChanged(New ListChangedEventArgs(ListChangedType.Reset, -1))
    End If

    m_isFiltering = False
End Sub
```

The *RemoveFilterInternal* function restores the original list if the list is currently not being sorted, or it copies the original list and then calls *Sort* if the list is currently being sorted.

The *Filter* property calls *ApplyFilter* if no filter has been applied yet, as follows.

#### C#

```
void ApplyFilter()
{
    if (m_isFiltering)
    {
        return;
    }

    if (m_originalList == null)
    {
        m_originalList = m_data;
        m_data = new List<T>();
    }

    m_isFiltering = true;

    ReapplyFilter();
}
```

#### Visual Basic

```
Sub ApplyFilter()
    If (m_isFiltering) Then
        Return
    End If

    If (m_originalList Is Nothing) Then
        m_originalList = m_data
        m_data = New List(Of T)()
    End If

    m_isFiltering = True

    ReapplyFilter()
End Sub
```

*ApplyFilter* saves the original list and creates an empty list if the list is currently not being sorted. The empty list is filled in by the *ReapplyFilter* method.

The *ReapplyFilter* method traverses the original list and checks the *IsFiltered* method to see whether the item has been filtered.

**C#**

```
bool IsFiltered(T item)
{
    if (_filterRoot == null)
    {
        return false;
    }

    return _filterRoot.Eval(item);
}

void ReapplyFilter()
{
    if (!IsFiltering)
    {
        return;
    }

    m_data.Clear();
    foreach (T item in m_originalList)
    {
        if (IsFiltered(item))
        {
            m_data.Add(item);
        }
    }

    if (IsSorted)
    {
        Sort();
    }

    OnListChanged(new ListChangedEventArgs(ListChangedType.Reset, -1));
}
```

**Visual Basic**

```
Function IsFiltered(ByVal item As T) As Boolean
    If (_filterRoot Is Nothing) Then
        Return False
    End If

    Return _filterRoot.Eval(item)
End Function

Sub ReapplyFilter()
    If (Not m_isFiltering) Then
        Return
    End If
```

```

m_data.Clear()
For Each item As T In m_originalList
    If (IsFiltered(item)) Then
        m_data.Add(item)
    End If
Next

If (IsSorted) Then
    Sort()
End If

OnListChanged(New ListChangedEventArgs(ListChangedType.Reset, -1))
End Sub

```

**RemoveFilter** This method removes the current filter. This is the equivalent of writing *Filter = String.Empty*.

#### C#

```

public void RemoveFilter()
{
    if (!SupportsAdvancedSorting)
    {
        throw new NotSupportedException();
    }

    Filter = String.Empty;
}

```

#### Visual Basic

```

Public Sub RemoveFilter() Implements IBindingListView.RemoveFilter
    If (Not SupportsAdvancedSorting) Then
        Throw New NotSupportedException()
    End If

    Filter = String.Empty
End Sub

```

## Adding Advanced Sorting

You can support advanced sorting through the *SupportsAdvancedSorting* and *SortDescriptions* properties as well as the *ApplySort* method. The *ApplySort* and *RemoveSort* methods as well as the *SortDirection* and *SortProperty* properties in the *IBindingList* implementation need to be modified to support both simple and advanced sorting.

**SupportsAdvancedSorting** This property states whether the list supports advanced sorting.

**SortDescriptions** This property contains the list of currently sorted descriptors.

**Modifications to *IBindingList*** *ApplySort* must be modified to add the specified sort properties to *m\_sortDescriptors*.

**C#**

```
public void ApplySort(PropertyDescriptor property, ListSortDirection direction)
{
    if (!SupportsSorting)
    {
        throw new NotSupportedException();
    }

    SaveUnsortedList(false);

    m_sortDescriptors.Clear();
    m_sortDescriptors.Add(new ListSortDescription(property, direction));
    IsSorted = true;
    Sort();
}
```

**Visual Basic**

```
Public Sub ApplySort(ByVal sorts As ListSortDescriptionCollection) _
    Implements IBindingListView.ApplySort
    If (Not SupportsAdvancedSorting) Then
        Throw New NotSupportedException()
    End If

    SaveUnsortedList(True)

    m_sortDescriptors.Clear()
    For Each sort As ListSortDescription In sorts
        m_sortDescriptors.Add(sort)
    Next
    m_isSorted = True
    Sort()
End Sub
```

The *RemoveSort* method restores the original list if filtering is not enabled and notifies the bound item of the list change.

**C#**

```
public void RemoveSort()
{
    if (!SupportsSorting)
    {
        throw new NotSupportedException();
    }
    m_sortDescriptors.Clear();
    IsSorted = false;
    if (m_originalList != null)
    {
        m_data = m_originalList;
        m_originalList = null;
    }

    if (IsFiltering)
```

```

    {
        ReapplyFilter();
    }

    if (SupportsSearching)
    {
        ReIndex();
    }
    OnListChanged(new ListChangedEventArgs(ListChangedType.Reset, -1));
}

```

### Visual Basic

```

Public Sub RemoveSort() Implements IBindingList.RemoveSort
    If (Not SupportsSorting) Then
        Throw New NotSupportedException()
    End If
    m_sortDescriptors.Clear()
    m_isSorted = False
    If (m_originalList IsNot Nothing) Then
        m_data = m_originalList
        m_originalList = Nothing
    End If

    If (IsFiltering) Then
        ReapplyFilter()
    End If

    If (SupportsSearching) Then
        ReIndex()
    End If
    OnListChanged(New ListChangedEventArgs(ListChangedType.Reset, -1))
End Sub

```

The *SortDirection* and *SortProperty* properties return the first item in *m\_sortDescriptors* if *m\_sortDescriptors* contains only one item.

### C#

```

public ListSortDirection SortDirection
{
    get
    {
        if (SupportsSorting)
        {
            if (m_sortDescriptors.Count == 1)
            {
                return m_sortDescriptors[0].SortDirection;
            }
            return ListSortDirection.Ascending;
        }

        throw new NotSupportedException();
    }
}

```

```

public PropertyDescriptor SortProperty
{
    get
    {
        if (SupportsSorting)
        {
            if (m_sortDescriptors.Count == 1)
            {
                return m_sortDescriptors[0].PropertyDescriptor;
            }
            return null;
        }

        throw new NotSupportedException();
    }
}

```

### Visual Basic

```

Public ReadOnly Property SortDirection() As ListSortDirection _
    Implements IBindingList.SortDirection
    Get
        If (SupportsSorting) Then
            If (m_sortDescriptors.Count = 1) Then
                Return m_sortDescriptors(0).SortDirection
            End If
            Return ListSortDirection.Ascending
        End If

        Throw New NotSupportedException()
    End Get
End Property

Public ReadOnly Property SortProperty() As PropertyDescriptor _
    Implements IBindingList.SortProperty
    Get
        If (SupportsSorting) Then
            If (m_sortDescriptors.Count = 1) Then
                Return m_sortDescriptors(0).PropertyDescriptor
            End If
            Return Nothing
        End If

        Throw New NotSupportedException()
    End Get
End Property

```

The *Sort* method must also be modified to traverse through each item in *m\_sortDescriptors*.

### C#

```

void Sort()
{
    if (m_sortDescriptors.Count <= 0)
    {
        return;
    }
}

```

```

m_data.Sort(
    new LambdaComparer<T>
    (
        (x, y) =>
        {

            for (int i = 0; i < m_sortDescriptors.Count; ++i)
            {
                var sd = m_sortDescriptors[i];

                object xValue = sd.PropertyDescriptor.GetValue(x);
                object yValue = sd.PropertyDescriptor.GetValue(y);

                int result = 0;

                if (sd.SortDirection == ListSortDirection.Descending)
                {
                    result = System.Collections.Comparer.Default.Compare
                        (xValue, yValue) * -1;
                }
                else
                {
                    result = System.Collections.Comparer.Default.Compare
                        (xValue, yValue);
                }

                if (result != 0 || i == m_sortDescriptors.Count - 1)
                {
                    return result;
                }
            }

            System.Diagnostics.Debug.Assert(false);
            return 0;
        }
    );
}

if (SupportsSearching)
{
    ReIndex();
}

OnListChanged(new ListChangedEventArgs(ListChangedType.Reset, -1));
}

```

**Visual Basic**

```

Function Compare(ByVal x As T, ByVal y As T) As Integer

For i As Integer = 0 To m_sortDescriptors.Count - 1
    Dim sd = m_sortDescriptors(i)

    Dim xValue As Object = sd.PropertyDescriptor.GetValue(x)
    Dim yValue = sd.PropertyDescriptor.GetValue(y)

```

```
Dim result As Integer = 0

If (sd.SortDirection = ListSortDirection.Descending) Then
    result = System.Collections.Comparer.Default.Compare(xValue, yValue) * -1
Else
    result = System.Collections.Comparer.Default.Compare(xValue, yValue)
End If

If (result <> 0 Or i = m_sortDescriptors.Count - 1) Then
    Return result
End If
Next

System.Diagnostics.Debug.Assert(False)
Return 0
End Function

Sub Sort()
    If (m_sortDescriptors.Count <= 0) Then
        Return
    End If

    m_data.Sort(AddressOf Compare)

    If (SupportsSearching) Then
        ReIndex()
    End If

    OnListChanged(New ListChangedEventArgs(ListChangedType.Reset, -1))
End Sub
```

**ApplySort** This method sorts on all of the items in sorts.

### C#

```
public void ApplySort(ListSortDescriptionCollection sorts)
{
    if (!SupportsAdvancedSorting)
    {
        throw new NotSupportedException();
    }

    SaveUnsortedList(true);

    m_sortDescriptors.Clear();
    foreach (ListSortDescription sort in sorts)
    {
        m_sortDescriptors.Add(sort);
    }
    IsSorted = true;
    Sort();
}
```

**Visual Basic**

```

Public Sub ApplySort( _
    ByVal [property] As PropertyDescriptor, _
    ByVal direction As ListSortDirection _ 
) Implements IBindingList.ApplySort
If (Not SupportsSorting) Then
    Throw New NotSupportedException()
End If

SaveUnsortedList(False)

m_sortDescriptors.Clear()
m_sortDescriptors.Add(New ListSortDescription([property], direction))
m_isSorted = True
Sort()
End Sub

```

The *SaveUnsortedList* method saves the original list if the list is not currently being sorted.

**C#**

```

void SaveUnsortedList(bool advance)
{
    if (!IsSorted)
    {
        if (m_originalList == null)
        {
            m_originalList = m_data;
            m_data = new List<T>(m_originalList);
        }
    }
}

```

**Visual Basic**

```

Sub SaveUnsortedList(ByVal advance As Boolean)
    If (Not IsSorted) Then
        If (m_originalList Is Nothing) Then
            m_originalList = m_data
            m_data = New List(Of T)(m_originalList)
        End If
    End If
End Sub

```

## Using the *BindingList(T)* Class

*BindingList(T)* provides a generic implementation of *IBindingList* that simplifies the creation of a custom implementation of the *IBindingList* interface. *BindingList(T)* implements *IBindingList* and provides generic support for bounded collections. To override the default implementation of *BindingList(T)*, look for the method or property with the word *Core* appended to it. For example, to implement *RemoveSort*, you need to override the *RemoveSortCore* method. See the section “Implementing the *IBindingList* Interface” earlier in this chapter for more information about each method and property.

## Using the *BindingSource* Class

The *BindingSource* class provides search and sort capability to a data source that implements *IBindingList*. If the data source you plan to use doesn't implement *IBindingList*, and you want bound items to see your changes, you need to call the appropriate methods in the *BindingSource* class rather than on the data source.



**Warning** According to the Microsoft documentation at [http://msdn.microsoft.com/en-us/library/system.windows.forms.bindingsource\(v=VS.100\).aspx#Y8235](http://msdn.microsoft.com/en-us/library/system.windows.forms.bindingsource(v=VS.100).aspx#Y8235), the *DataSource* property value of *BindingSource* should be changed on the user interface (UI) thread to ensure that the UI reflects the changes.

To use *BindingSource*, create an instance of the *BindingSource* class and assign the *DataSource* property to your collection.

### C#

```
List<int> items = new List<int>();  
  
// Populate items  
  
BindingSource source = new BindingSource();  
source.DataSource = items;
```

### Visual Basic

```
Dim items As New List(Of Integer())  
  
' Populate items  
  
Dim source As New BindingSource()  
source.DataSource = items
```

Then set the control to the *BindingSource* as follows.

### C#

```
dataGridView1.DataSource = source;
```

### Visual Basic

```
dataGridView1.DataSource = source
```

You need to call the *Add*, *Clear*, *Insert*, *Remove*, and *RemoveAt* methods in *BindingSource* instead of the data source if your data source doesn't implement *IBindingList*, as follows.

### C#

```
source.Add(1);  
source.RemoveAt(0);
```

### Visual Basic

```
source.Add(1)  
source.RemoveAt(0)
```

## Understanding the Sample Code

The sample project *Driver*, located in the Chapter 10\CS\Driver folder, or the Chapter 10\VB\Driver folder, contains examples of using the binding interfaces and objects for collections with a *ComboBox*, *DataGridView*, and *ListBox*.

### Binding with the *ComboBox* Control

The *ComboBoxBinding* form demonstrates how to use the binding classes with a combo box, which is created using the *ComboBox* class in Windows Forms. An object that implements *IBindingList* can bind to a combo box by using the following code.

#### C#

```
comboBox1.DataSource = m_datasource;
comboBox1.DisplayMember = "Name";
comboBox1.ValueMember = "Id";
```

#### Visual Basic

```
comboBox1.DataSource = m_datasource
comboBox1.DisplayMember = "Name"
comboBox1.ValueMember = "Id"
```

The Update panel shows the currently selected item in the combo box. You can update the selected item by clicking on the Update Item button. The combo box is automatically updated to reflect the updated item. Because each item implements *INotifyPropertyChanged*, bound controls receive notifications of property changes when items are updated with the following code.

#### C#

```
m_showing.Name = NameTextBox.Text;
m_showing.Website = WebsiteTextBox.Text;
```

#### Visual Basic

```
m_showing.Name = NameTextBox.Text
m_showing.Website = WebsiteTextBox.Text
```

You can use the Add panel to add new items to the combo box. You add items to the combo box by clicking the Add Item button. The combo box is automatically updated to reflect the newly added item. The code for adding an item is as follows.

#### C#

```
Company company = new Company();

company.Id = int.Parse(AddIdTextBox.Text);
company.Name = AddNameTextBox.Text;
company.Website = AddWebsiteTextBox.Text;

m_datasource.Add(company);
```

**Visual Basic**

```
Dim company As New Company()

company.Id = Integer.Parse(AddIdTextBox.Text)
company.Name = AddNameTextBox.Text
company.Website = AddWebsiteTextBox.Text

m_datasource.Add(company)
```

You can remove the selected item in the combo box by clicking the Remove Item button. The following shows the code for removing an item.

**C#**

```
if (comboBox1.SelectedIndex >= 0)
{
    m_datasource.RemoveAt(comboBox1.SelectedIndex);
}
```

**Visual Basic**

```
If (comboBox1.SelectedIndex >= 0) Then
    m_datasource.RemoveAt(comboBox1.SelectedIndex)
End If
```

## Binding with the *ListBox* Control

The *ListBoxBinding* form demonstrates how to use the binding classes with a list box, which is created using the *ListBox* class in Windows Forms. An object that implements *IBindingList* can bind to a list box by using the following code.

**C#**

```
listBox1.DataSource = m_datasource;
listBox1.DisplayMember = "Name";
```

**Visual Basic**

```
listBox1.DataSource = m_datasource
listBox1.DisplayMember = "Name"
```

The Update panel shows the currently selected item in the list box. You update the selected item by clicking the Update Item button. The list box is automatically updated to reflect the updated item. Because each item implements *IPropertyChanged*, bound controls receive notifications of property changes when items are updated with the following code.

**C#**

```
m_showing.Name = NameTextBox.Text;
m_showing.Website = WebsiteTextBox.Text;
```

**Visual Basic**

```
m_showing.Name = NameTextBox.Text
m_showing.Website = WebsiteTextBox.Text
```

You can use the Add panel to add new items to the list box. You add items to the list box by clicking the Add Item button. The list box is automatically updated to reflect the newly added item. The code for adding an item is as follows.

**C#**

```
Company company = new Company();

company.Id = int.Parse(AddIdTextBox.Text);
company.Name = AddNameTextBox.Text;
company.Website = AddWebsiteTextBox.Text;

m_datasource.Add(company);
```

**Visual Basic**

```
Dim company As New Company()

company.Id = Integer.Parse(AddIdTextBox.Text)
company.Name = AddNameTextBox.Text
company.Website = AddWebsiteTextBox.Text

m_datasource.Add(company)
```

You can remove the selected item in the list box by clicking the Remove Item button. The following shows the code for removing an item.

**C#**

```
if (comboBox1.SelectedIndex >= 0)
{
    m_datasource.RemoveAt(listBox1.SelectedIndex);
}
```

**Visual Basic**

```
If (comboBox1.SelectedIndex >= 0) Then
    m_datasource.RemoveAt(listBox1.SelectedIndex)
End If
```

## Binding with the *DataGridView* Control and *IBindingList*

The *DataGridViewBinding* form demonstrates how to use the binding classes with a data grid, which is created using the *DataGridView* class in Windows Forms. An object that implements *IBindingList* can bind to a *DataGridView* by using the following code.

**C#**

```
m_datasource = DL.GetDataSource();
dataGridView1.DataSource = m_datasource;
```

**Visual Basic**

```
m_datasource = DL.GetDataSource()
dataGridView1.DataSource = m_datasource
```

Items can be added and removed from the data grid by using the same code that is in the *ComboBoxBinding* and *ListBoxBinding* forms. In fact, both of the forms use the same data source instance in the sample code as the *DataGridViewBinding* form, so updating the data source in the *ComboBoxBinding* or *ListBoxBinding* form also updates the *DataGridViewBinding* form.

Items can be searched by entering a search string in the search box, selecting a property to search on, and clicking the Search button. A message box will appear stating the row the item was found in. You can also test indexing by selecting the properties you want to index on and then searching on the indexed property. The code for searching is as follows.

**C#**

```
if (string.IsNullOrEmpty(SearchTextBox.Text) || SearchPropertyComboBox.SelectedIndex < 0)
{
    return;
}

var pd = SearchPropertyComboBox.SelectedItem as PropertyDescriptor;

int found = -1;

try
{
    found = m_datasource.Find(pd, pd.Converter.ConvertFromString(SearchTextBox.Text));

    if (found >= 0)
    {
        MessageBox.Show(string.Format("Found '{0}' at index {1}",
                                     SearchTextBox.Text, found));
    }
    else
    {
        MessageBox.Show(string.Format("Didn't find '{0}'", SearchTextBox.Text));
    }
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

**Visual Basic**

```
If (String.IsNullOrEmpty(SearchTextBox.Text) Or SearchPropertyComboBox.SelectedIndex < 0)
Then
    Return
End If

Dim pd = TryCast(SearchPropertyComboBox.SelectedItem, PropertyDescriptor)

Dim found As Integer = -1
```

```

Try
    found = m_datasource.Find(pd, pd.Converter.ConvertFromString(SearchTextBox.Text))

    If (found >= 0) Then
        MessageBox.Show(String.Format("Found '{0}' at index {1}", _
            SearchTextBox.Text, found))
    Else
        MessageBox.Show(String.Format("Didn't find '{0}'", SearchTextBox.Text))
    End If
Catch ex As Exception
    MessageBox.Show(ex.Message)
End Try

```

## Binding with the *DataGridView* Control and *IBindingListView*

The *DataGridViewAdvanceBinding* form is the *DataGridViewBinding* form with additional support for the *IBindingListView* interface. The *IBindingListView* interface allows users to filter the results and sort on multiple columns. The following code shows how to bind *IBindingListView* object to the data grid control.

### C#

```

m_datasource = DL.GetDataSourceView();
m_binding = New BindingSource();
m_binding.DataSource = m_datasource;
dataGridView1.DataSource = m_binding;

```

### Visual Basic

```

m_datasource = DL.GetDataSourceView()
m_binding = New BindingSource()
m_binding.DataSource = m_datasource
dataGridView1.DataSource = m_binding

```

Users can filter items by entering the filter string into the Filter text box and clicking the Filter button. The following code shows how to filter the *IBindingListView* object.

### C#

```
m_datasource.Filter = FilterTextBox.Text;
```

### Visual Basic

```
m_datasource.Filter = FilterTextBox.Text
```

The *IBindingListView* collection can be sorted on multiple properties by using the *Sort* property on the *BindingSource*. To do this in the UI, enter the sort text into the Sort text box, and click the Sort button. The code for doing this is as follows.

### C#

```
m_binding.Sort = SortTextBox.Text;
```

### Visual Basic

```
m_binding.Sort = SortTextBox.Text
```

## Binding with the *BindingSource* Object

The *BindingSourceBinding* form demonstrates how to perform two-way binding on a collection that doesn't implement *IBindingList*. A collection that doesn't implement *IBindingList* can bind to a list box by using the following code.

C#

```
m_source = new BindingSource();
m_source.DataSource = new List<Company>(DL.GetData());
listBox1.DataSource = m_source;
listBox1.DisplayMember = "Name";
```

Visual Basic

```
m_source = New BindingSource()
m_source.DataSource = New List(Of Company)(DL.GetData())
listBox1.DataSource = m_source
listBox1.DisplayMember = "Name"
```

The Update panel shows the currently selected item in the list box. You can update the selected item by clicking the Update Item button. The list box is automatically updated to reflect the updated item. Because each item implements *INotifyPropertyChanged*, items are updated with the following code.

C#

```
m_showing.Name = NameTextBox.Text;
m_showing.Website = WebsiteTextBox.Text;
```

Visual Basic

```
m_showing.Name = NameTextBox.Text
m_showing.Website = WebsiteTextBox.Text
```

You can use the Add panel to add new items to the list box. You add items to the list box by clicking the Add Item button. The list box updates automatically to reflect the newly added item. The item needs to be added using the *BindingSource* instead of the *List(T)* for the list box to see the changes, because *List(T)* doesn't implement *IBindingList*.

C#

```
Company company = new Company();

company.Id = int.Parse(AddIdTextBox.Text);
company.Name = AddNameTextBox.Text;
company.Website = AddWebsiteTextBox.Text;

m_source.Add(company);
```

**Visual Basic**

```
Dim company As New Company()  
  
company.Id = Integer.Parse(AddIdTextBox.Text)  
company.Name = AddNameTextBox.Text  
company.Website = AddWebsiteTextBox.Text  
  
m_source.Add(company)
```

You can remove the selected item in the list box by clicking the Remove Item button. The following shows the code for removing an item. The item needs to be removed using the *BindingSource* instead of the *List(T)* for the list box to see the changes, because *List(T)* doesn't implement *IBindingList*.

**C#**

```
if (comboBox1.SelectedIndex >= 0)  
{  
    m_source.RemoveAt(listBox1.SelectedIndex);  
}
```

**Visual Basic**

```
If (comboBox1.SelectedIndex >= 0) Then  
    m_source.RemoveAt(listBox1.SelectedIndex)  
End If
```

## Summary

In this chapter, you saw how to bind collections to controls used in Windows Forms. You saw how to have two-way bound controls by using the *IBindingList* interface. You also saw that you can use the *IBindingList* interface to sort and search, and use the *IBindingListView* interface to do advanced filtering and sorting.

## Chapter 11

# Using Collections with WPF and Silverlight Controls

After completing this chapter, you will be able to

- Use and implement the *INotifyCollectionChanged* interface.
- Use the *ObservableCollection(T)* class.
- Use the *ICollectionView* and *CollectionView* derived classes.
- Bind collections to WPF and Silverlight controls.

## ***INotifyCollectionChanged* Overview**

The *INotifyCollectionChanged* interface is an interface that exposes a *CollectionChanged* event. The *CollectionChanged* event is raised whenever the collection is changed, such as when it is cleared, removed, or added to. The *INotifyCollectionChanged* interface is implemented by the *ObservableCollection(T)* class, which you learn about later in this chapter.

## **Implementing the *INotifyCollectionChanged* Interface**

This section walks you through implementing the *INotifyCollectionChange* interface by implementing a class named *NotificationList(T)*. *NotificationList(T)* also implements the *INotifyPropertyChanged* interface to inform the bound object of whenever the *Count* or *Item[]* property changes.



**Note** To see the final code, open the DevGuideToCollections project, in either the Chapter 11\CS\DevGuideToCollections folder or the Chapter 11\VB\DevGuideToCollections folder. The implementation of each interface implemented by *NotificationList(T)* is broken out into separate files to make it easier to follow. You'll find the implementation of *NotifyCollectionChanged* in the language-specific files *NotificationList.CollectionChanged.cs* and *NotificationList.CollectionChanged.vb*. You can review those files for the full implementation of the properties and methods discussed later in this section. Chapter 6, ".NET Collection Interfaces," and Chapter 8, "Using Threads with Collections," provide information about how to implement the other interfaces. *NotificationList(T)* is the *WinFormsBindingList(T)* class you created in Chapter 10, "Using Collections with Windows Form Controls," modified to support the *INotifyCollectionChanged* interface.

The *ListChanged* event must be raised whenever the list is modified. The list is modified whenever the *Add*, *Clear*, *Insert*, *Remove*, or *RemoveAt* method is called, as well as the *Item* set property. The easiest way to raise the *ListChanged* event when the list is modified is by creating *OnXXX* and *OnXXXComplete* methods like those discussed in the *CollectionBase* section in Chapter 5, "Generic and Support Collections." Each method except the *Add* method has *OnXXX* and *OnXXXComplete* methods. The *Add* and *Insert* methods use the *OnInsert* and *OnInsertComplete* methods; the *RemoveAt* and *Remove* methods use the *OnRemove* and *OnRemoveComplete* methods. The *OnXXX* method is called at the beginning of the operation and the *OnXXXComplete* method is called at the end of the method. The following code shows how the code in *ArrayEx(T)* is modified to support handling the *OnXXX* and *OnXXXComplete* methods.

C#

```
public void Add(T item)
{
    OnInsert(item, Count);
    InnerList.Add(item);
    OnInsertComplete(item, Count - 1);
}

public void Clear()
{
    var removed = ToArray();
    OnClear(removed);
    InnerList.Clear();
    OnClearComplete(removed);
}

public bool Remove(T item)
{
    int index = IndexOf(item);

    if (index >= 0)
    {
        OnRemove(item, index);

        InnerList.Remove(item);

        OnRemoveComplete(item, index);
    }

    return index >= 0;
}

public void RemoveAt(int index)
{
    if (index < 0 || index >= Count)
```

```
{  
    // Item has already been removed.  
    return;  
}  
  
T item = InnerList[index];  
  
OnRemove(item, index);  
  
InnerList.RemoveAt(index);  
  
OnRemoveComplete(item, index);  
}  
  
public void Insert(int index, T item)  
{  
    OnInsert(item, index);  
  
    InnerList.Insert(index, item);  
  
    OnInsertComplete(item, index);  
}  
  
public T this[int index]  
{  
    set  
    {  
        T oldValue = InnerList[index];  
  
        OnSet(index, oldValue, value);  
  
        InnerList[index] = value;  
  
        OnSetComplete(index, oldValue, value);  
    }  
}
```

### Visual Basic

```
Public Sub Add(ByVal item As T) Implements ICollection(Of T).Add  
    OnInsert(item, Count)  
    InnerList.Add(item)  
    OnInsertComplete(item, Count - 1)  
End Sub  
  
Public Sub Clear() Implements ICollection(Of T).Clear, IList.Clear  
    Dim removed As T() = ToArray()  
    OnClear(removed)  
    InnerList.Clear()  
    OnClearComplete(removed)  
End Sub
```

```
Public Function Remove(ByVal item As T) As Boolean Implements ICollection(Of T).Remove
    Dim index As Integer = IndexOf(item)

    If (index >= 0) Then
        OnRemove(item, index)

        InnerList.Remove(item)

        OnRemoveComplete(item, index)
    End If

    Return index >= 0
End Function

Public Sub RemoveAt(ByVal index As Integer) Implements IList(Of T).RemoveAt, IList.RemoveAt
    If (index < 0 Or index >= Count) Then
        ' Item has already been removed.
        Return
    End If

    Dim item As T = InnerList(index)

    OnRemove(item, index)

    InnerList.RemoveAt(index)

    OnRemoveComplete(item, index)
End Sub

Public Sub Insert(ByVal index As Integer, ByVal item As T) Implements IList(Of T).Insert
    OnInsert(item, index)

    InnerList.Insert(index, item)

    OnInsertComplete(item, index)
End Sub

Default Public Property Item(ByVal index As Integer) As T Implements IList(Of T).Item
    Set(ByVal value As T)
        Dim oldValue As T = InnerList(index)

        OnSet(index, oldValue, value)

        InnerList(index) = value

        OnSetComplete(index, oldValue, value)
    End Set
End Property
```

The *OnXXX* methods are not needed to implement the *INotifyCollectionChange* interface, but you might find some uses for them later. For now, they remain empty, as in the following example.

**C#**

```
void OnSet(int index, T oldValue, T newValue)
{
}

void OnClear(T[] itemsRemoved)
{
}

void OnInsert(T item, int index)
{
}

void OnRemove(T item, int index)
{
}
```

**Visual Basic**

```
Sub OnSet(ByVal index As Integer, ByVal oldValue As T, ByVal newValue As T)

End Sub

Sub OnClear(ByVal itemsRemoved As T())

End Sub

Sub OnInsert(ByVal item As T, ByVal index As Integer)

End Sub

Sub OnRemove(ByVal item As T, ByVal index As Integer)

End Sub
```

However, each of the *OnXXXComplete* methods needs to raise the appropriate event for the *XXX* operation. To simplify some of the coding, you can create two helper methods for raising the *PropertyChanged* and *CollectionChanged* events, as follows.

**C#**

```
void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

void OnCollectionChanged(NotifyCollectionChangedEventArgs e)
{
    if (CollectionChanged != null)
    {
        CollectionChanged(this, e);
    }
}
```

**Visual Basic**

```
Sub OnPropertyChanged(ByVal propertyName As String)
    RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(propertyName))
End Sub

Sub OnCollectionChanged(ByVal e As NotifyCollectionChangedEventArgs)
    RaiseEvent CollectionChanged(Me, e)
End Sub
```

Each *OnXXXComplete* method calls the *OnPropertyChanged* and *OnCollectionChanged* methods.

**C#**

```
void OnSetComplete(int index, T oldValue, T newValue)
{
    OnPropertyChanged("Item[]");
    OnCollectionChanged(new NotifyCollectionChangedEventArgs
        (NotifyCollectionChangedAction.Replace, oldValue, newValue, index));
}

void OnClearComplete(T[] itemsRemoved)
{
    OnPropertyChanged("Count");
    OnPropertyChanged("Item[]");

    object[] items = null;

    if (itemsRemoved != null)
    {
        items = new object[itemsRemoved.Length];

        for (int i = 0; i < itemsRemoved.Length; ++i)
        {
            items[i] = itemsRemoved[i];
        }
    }

    OnCollectionChanged(new NotifyCollectionChangedEventArgs(items));
}

void OnInsertComplete(T item, int index)
{
    OnPropertyChanged("Count");
    OnPropertyChanged("Item[]");
    OnCollectionChanged(new NotifyCollectionChangedEventArgs
        (NotifyCollectionChangedAction.Add, item, index));
}

void OnRemoveComplete(T item, int index)
{
    OnPropertyChanged("Count");
    OnPropertyChanged("Item[]");
    OnCollectionChanged(new NotifyCollectionChangedEventArgs
        (NotifyCollectionChangedAction.Remove, item, index));
}
```

### Visual Basic

```
Sub OnSetComplete(ByVal index As Integer, ByVal oldValue As T, ByVal newValue As T)
    OnPropertyChanged("Item[]")
    OnCollectionChanged(New NotifyCollectionChangedEventArgs _
        (NotifyCollectionChangedAction.Replace, oldValue, newValue, index))
End Sub

Sub OnClearComplete(ByVal itemsRemoved As T())
    OnPropertyChanged("Count")
    OnPropertyChanged("Item[]")

    Dim items As Object() = Nothing

    If (itemsRemoved IsNot Nothing) Then
        Dim tmp As List(Of Object) = New List(Of Object)

        For i As Integer = 0 To itemsRemoved.Length - 1
            tmp.Add(itemsRemoved(i))
        Next

        items = tmp.ToArray()
    End If

    OnCollectionChanged(New NotifyCollectionClearedEventArgs(items))
End Sub

Sub OnInsertComplete(ByVal item As T, ByVal index As Integer)
    OnPropertyChanged("Count")
    OnPropertyChanged("Item[]")
    OnCollectionChanged(New NotifyCollectionChangedEventArgs _
        (NotifyCollectionChangedAction.Add, item, index))
End Sub

Sub OnRemoveComplete(ByVal item As T, ByVal index As Integer)
    OnPropertyChanged("Count")
    OnPropertyChanged("Item[]")
    OnCollectionChanged(New NotifyCollectionChangedEventArgs _
        (NotifyCollectionChangedAction.Remove, item, index))
End Sub
```

Because most collection operations result in the size and contents of the collection changing, collection change operations also notify any bound objects of the *Count* and *Item[]* changing.

## Notifying the User of Cleared Items

You may have noticed that the *Clear* methods put the contents of the list in an array before calling the *OnClear* and *OnClearComplete* methods, as shown in the following code.

### C#

```
public void Clear()
{
    var removed = ToArray();
    OnClear(removed);
    InnerList.Clear();
    OnClearComplete(removed);
}
```

### Visual Basic

```
Public Sub Clear() Implements ICollection(Of T).Clear, IList.Clear
    Dim removed As T() = ToArray()
    OnClear(removed)
    InnerList.Clear()
    OnClearComplete(removed)
End Sub
```

The *OnClearComplete* method then raises the *CollectionChange* event with the items that are removed, as follows.

### C#

```
OnCollectionChanged(new NotifyCollectionChangedEventArgs(items));
```

### Visual Basic

```
OnCollectionChanged(new NotifyCollectionChangedEventArgs(items))
```

The advantage of doing this is that it allows any attached objects to unregister from the objects being cleared. Because the event is raised after the collection is cleared, the receiver wouldn't have access to the cleared objects otherwise.

The *NotifyCollectionChangedEventArgs* parameter doesn't support passing in a collection of items with the *Reset* action. To get around this, a custom class called *NotifyCollectionClearedEventArgs* is created that is derived from the *NotifyCollectionChangedEventArgs* class. The *NotifyCollectionClearedEventArgs* class exposes the *ClearedItems* property so that any attached delegates can retrieve the cleared items. *NotifyCollectionClearedEventArgs* is defined as follows.

**C#**

```
public class NotifyCollectionClearedEventArgs : NotifyCollectionChangedEventArgs
{
    object[] m_clearedItems;

    public NotifyCollectionClearedEventArgs(object[] clearedItems)
        : base(NotifyCollectionChangedAction.Reset)
    {
        m_clearedItems = clearedItems;
    }

    public object[] ClearedItems
    {
        get { return m_clearedItems; }
    }
}
```

**Visual Basic**

```
Public Class NotifyCollectionClearedEventArgs
    Inherits NotifyCollectionChangedEventArgs

    Private m_clearedItems As Object()

    Public Sub New(ByVal clearedItems As Object())
        MyBase.new(NotifyCollectionChangedAction.Reset)
        m_clearedItems = clearedItems
    End Sub

    Public ReadOnly Property ClearedItems() As Object()
        Get
            Return m_clearedItems
        End Get
    End Property
End Class
```

## Resolving Problems with the Recursive Collection Change Event

In Chapter 8, you learned that you have to be careful when changing collections within locks. You also must be careful when changing collections within notification events. If you change the collection that the event was raised on or trigger events that cause the collection to change, your code might run in a recursive loop until your application crashes or runs out of memory. Take a look at the following example.

**C#**

```

ObservableCollection<int> numbersToProcess = new ObservableCollection<int>();
ObservableCollection<int> processedNumbers = new ObservableCollection<int>();

numbersToProcess.CollectionChanged += 
    (sender, e) =>
{
    if (e.Action == 
        System.Collections.Specialized.NotifyCollectionChangedEventArgs.Add)
    {
        processedNumbers.Add((int)e.NewItems[0]);
        numbersToProcess.RemoveAt(e.NewStartingIndex);
    }
};

processedNumbers.CollectionChanged += 
    (sender, e) =>
{
    if (e.Action == System.Collections.Specialized.NotifyCollectionChangedEventArgs.Add)
    {
        int value = (int)e.NewItems[0];

        if (value % 2 != 0)
        {
            // Not able to process odd numbers yet
            numbersToProcess.Add(value);
        }
    }
};

numbersToProcess.Add(1);

```

**Visual Basic**

```

Private numbersToProcess As New ObservableCollection(Of Int32)()
Private processedNumbers As New ObservableCollection(Of Int32)()

Sub OnNumbersToProcessChanged(ByVal sender As Object, _
    ByVal e As NotifyCollectionChangedEventArgs)
    If (e.Action = System.Collections.Specialized.NotifyCollectionChangedEventArgs.Add) Then
        processedNumbers.Add(CType(e.NewItems(0), Int32))
        numbersToProcess.RemoveAt(e.NewStartingIndex)
    End If
End Sub

Sub OnProcessedNumbersChanged(ByVal sender As Object, _
    ByVal e As NotifyCollectionChangedEventArgs)
    If (e.Action = System.Collections.Specialized.NotifyCollectionChangedEventArgs.Add) Then
        Dim value As Int32 = CType(e.NewItems(0), Int32)

        If (value Mod 2 <> 0) Then
            ' Not able to process odd numbers yet
            numbersToProcess.Add(value)
        End If
    End If
End Sub

```

```
End If  
End Sub  
  
AddHandler numbersToProcess.CollectionChanged, AddressOf OnNumbersToProcessChanged  
AddHandler processedNumbers.CollectionChanged, AddressOf OnProcessedNumbersChanged  
  
numbersToProcess.Add(1)
```

The code simulates processing numbers as they are added to the collection by hooking to the *CollectionChanged* event of *numbersToProcess*. As numbers are added to the list, they are then added to *processedNumbers* to simulate the numbers being processed. The *process-Numbers* collection can process only numbers that are even; any odd numbers are added back to the *numbersToProcess* collection so that they can be processed later. This causes the *CollectionChanged* event to be raised on *numbersToProcess* again, and the operation happens over and over until a *StackOverflowException* exception is thrown. This example shows you how changing collections in events can cause some very hard-to-find bugs when the *CollectionChanged* event of *processedNumbers* or *numbersToProcess* triggers several other events that eventually do the same as the preceding code.

The solution for the preceding code is to not change the collection within its change event. For this simple scenario, you could solve it by checking to see whether *numbersToProcess* contains the item you are adding before you add it. This solution, however, probably wouldn't work in a real-world scenario. See the "Handling Recursive Collection Change Events" section later in this chapter to see what the *ObservableCollection(T)* class does in this case.

## ***ObservableCollection(T)* Overview**

The *ObservableCollection(T)* class is a generic collection that implements *INotifyCollection-Changed*. The class is perfect for data binding in Windows Presentation Foundation (WPF) and Microsoft Silverlight because it implements the *INotifyCollectionChanged* interface, unlike the *List(T)* class you learn about in Chapter 4, "Generic Collections." The *INotifyCollectionChanged* interfaces notify the bound control, such as the *ListBox*, *ListView*, and *TreeView*, of changes whenever the collection is modified.

## **Using the *ObservableCollection(T)* Class**

The *ObservableCollection(T)* class is used like the *List(T)* class. The only difference is that *ObservableCollection(T)* implements the *INotifyCollectionChanged* event, which allows you to register for collection change events, as shown in the following code.



**More Info** For information on how to use the *List(T)* class, see Chapter 4. For a brief overview on lists or arrays, see Chapter 1, "Understanding Collections: Arrays and Linked Lists."

**C#**

```
ObservableCollection<int> list = new ObservableCollection<int>();

list.CollectionChanged += (sender, e) =>
{
    switch (e.Action)
    {
        case System.Collections.Specialized.NotifyCollectionChangedEventArgs.Add:
            Console.WriteLine("{0} was added to the list", e.NewItems[0]);
            break;
        case System.Collections.Specialized.NotifyCollectionChangedEventArgs.Remove:
            Console.WriteLine("{0} was removed from the list", e.OldItems[0]);
            break;
        case System.Collections.Specialized.NotifyCollectionChangedEventArgs.Reset:
            Console.WriteLine("The list was cleared");
            break;
    }
};

list.Add(1);
list.Add(4);
list.Remove(1);
list.Clear();
```

**Visual Basic**

```
Sub OnCollectionChanged(ByVal sender As Object, ByVal e As NotifyCollectionChangedEventArgs)
    Select Case (e.Action)
        Case System.Collections.Specialized.NotifyCollectionChangedEventArgs.Add
            Console.WriteLine("{0} was added to the list", e.NewItems(0))
        Case System.Collections.Specialized.NotifyCollectionChangedEventArgs.Remove
            Console.WriteLine("{0} was removed from the list", e.OldItems(0))
        Case System.Collections.Specialized.NotifyCollectionChangedEventArgs.Reset
            Console.WriteLine("The list was cleared")
    End Select
End Sub

Dim list As ObservableCollection(Of Int32) = New ObservableCollection(Of Int32)()

AddHandler list.CollectionChanged, AddressOf OnCollectionChanged

list.Add(1)
list.Add(4)
list.Remove(1)
list.Clear()
```

**Output**

```
1 was added to the list
4 was added to the list
1 was removed from the list
The list was cleared
```



**Note** Unlike the *NotificationList(T)* that you created earlier in this chapter, the *ObservableCollection(T)* does not pass to you the items that were cleared in the *CollectionChanged* event when you call the *Clear* method. That can present a problem if you need to unregister from items that are being cleared.

## Handling Recursive Collection Change Events

Earlier in the chapter, you learned a bit about raising collection-changed events in the collection-change event you are currently in. The *ObservableCollection(T)* informs you of this operation by throwing an *InvalidOperationException* exception when this occurs. The class determines that your code is in a recursive call by incrementing a value before raising the *CollectionChanged* event and then decrementing the value after the call. This can be accomplished by doing the following.

### C#

```
int m_recursive;

void OnCollectionChanged(NotifyCollectionChangedEventArgs e)
{
    if (CollectionChanged != null)
    {
        try
        {
            ++m_recursive;
            CollectionChanged(this, e);
        }
        finally
        {
            --m_recursive;
        }
    }
}
```

### Visual Basic

```
Dim m_recursive As Int32

Sub OnCollectionChanged(ByVal e As NotifyCollectionChangedEventArgs)
    Try
        m_recursive += 1
        RaiseEvent CollectionChanged(this, e)
    Finally
        m_recursive -= 1
    End Try
End Sub
```

A check is then done before each list change operation such as *Clear*, *Remove*, *RemoveAt*, and *Add*, to see if *m\_recursive* is greater than 0. A value greater than 0 means that someone is currently in the *OnCollectionChanged* event. The invocation list of *CollectionChanged* is also checked to see if more than one delegate is registered to receive *CollectionChanged* events. This presents an issue with the preceding coding example because only one delegate is registered to receive *CollectionChanged* events.

#### C#

```
void CheckForReentry()
{
    if (m_recursive > 0 && CollectionChanged.GetInvocationList().Length > 1)
    {
        throw new InvalidOperationException();
    }
}
```

#### Visual Basic

```
Sub CheckForReentry()
    If (m_recursive > 0 And CollectionChanged.GetInvocationList().Length > 1) Then
        Throw New InvalidOperationException()
    End If
End Sub
```

Each collection change method checks for reentrance before doing any operations, as follows.

#### Pseudocode

```
... Method(...)
{
    CheckForReentry();

    ...

    OnPropertyChanged("Count");
    OnPropertyChanged("Item[]");

    OnCollectionChanged(...);
}
```

## IICollectionView and CollectionView Overview

The *IICollectionView* interface provides support for filtering, sorting, grouping, and selecting items in a collection. *CollectionView* is a class provided by the Microsoft .NET Framework, which implements the *IICollectionView* interface. The advantage of the class is that it allows you to filter, sort, and group the items in a collection without manipulating the underlying source collection if the *IBindingList* and *IBindingListView* interfaces that you learned about in Chapter 10 are implemented on the source collection.

The *IICollectionView* interface is designed to be implemented in a separate class, which allows you to have multiple objects that implement the *IICollectionView* interface attached to the same instance of a collection. You can use *CollectionView* for simple collections that only implement *IEnumerable*. For classes that implement *IBindingList* or *IList*, you might want to look at the *BindingListCollectionView* and *ListCollectionView* classes respectively.

## When to Use *BindingListCollectionView*

The *BindingListCollectionView* is an implementation of *IICollectionView* that can be used on objects that implement the *IBindingList* interface. Filtering does not work on the *BindingListCollectionView*. Also, sorting does not work if the bound list doesn't implement *ITypedList* and if *SupportsSorting* is set to *false*. For the example later in this chapter, you use a modified version of *WinFormsBindingList(T)* that implements *ITypedList*. The modified version contains the following implementation for *ITypedList*, as found on MSDN at <http://msdn.microsoft.com/en-us/library/ms404298.aspx>.

### C#

```
public PropertyDescriptorCollection GetItemProperties(PropertyDescriptor[] listAccessors)
{
    if (_properties == null)
    {
        // Only get the public properties marked with Browsable = true.
        PropertyDescriptorCollection pdc = TypeDescriptor.GetProperties(
            typeof(T), new Attribute[] { new BrowsableAttribute(true) });

        // Sort the properties.
        _properties = pdc.Sort();
    }

    return _properties;
}

// This method is only used in the design-time framework
// and by the obsolete DataGrid control.
public string GetListName(PropertyDescriptor[] listAccessors)
{
    return typeof(T).Name;
}
```

### Visual Basic

```
Public Function GetItemProperties(ByVal listAccessors As PropertyDescriptor()) As
    PropertyDescriptorCollection Implements ITypedList.GetItemProperties
    If (_properties Is Nothing) Then
        ' Only get the public properties marked with Browsable = true.
        Dim pdc As TypeDescriptor.GetProperties _
            (GetType(T), New Attribute() {New BrowsableAttribute(True)})
```

```

' Sort the properties.
m_properties = pdc.Sort()
End If

Return m_properties
End Function

' This method is only used in the design-time framework
' and by the obsolete DataGrid control.
Public Function GetListName(ByVal listAccessors As PropertyDescriptor()) As String _
    Implements ITypedList.GetListName
    Return GetType(T).Name
End Function

```

The *ITypedList* interface defines the properties to bind to when they differ from the public properties of the bound object. For the collection class, you want the properties of the object defined by *T* instead of *WinFormsBindingList(T)*.

Later in this chapter is an example that uses the *BindingListViewCollectionView*.

## When to Use *ListCollectionView*

The *ListCollectionView* is an implementation of *ICollectionView* that can be used on objects that implement the *IList* interface. The user interface (UI) is not notified of any collection changes unless *INotifyCollectionChanged* is implemented as well.

Later in the chapter, you will see an example using the *ListCollectionView*.

## Understanding the Sample Code

The sample project, located in the Chapter 11\CS\Driver folder or the Chapter 11\VB\Driver folder, contains examples of using bound collections with a *ComboBox*, *ListBox*, *ListView*, and *TreeView*. The sample code also shows how to use the *ICollectionView* interface.

## Binding with the *ComboBox* Control

The *ComboBoxBinding* window demonstrates how to use the binding classes with a combo box, which is created using the *ComboBox* class in WPF and Silverlight. An object that implements *IEnumerable* can bind to a combo box by using the following code.

C#

```

ExampleComboBox.ItemsSource = m_datasource;
ExampleComboBox.DisplayMemberPath = "Name";
ExampleComboBox.SelectedValuePath = "Id";

```

### Visual Basic

```
ExampleComboBox.ItemsSource = m_datasource  
ExampleComboBox.DisplayMemberPath = "Name"  
ExampleComboBox.SelectedValuePath = "Id"
```

However, the combo box does not receive dynamic updates unless the collection implements *INotifyCollectionChanged*. To have the combo box receive notifications, you use the *ObservableCollection(T)* class, which implements *INotifyCollectionChanged* as well as the *NotificationList(T)* you created earlier in this chapter.

The Update panel shows the currently selected item in the combo box. You update the selected item by clicking the Update Item button. The combo box is automatically updated to reflect the updated item. Because each item implements *INotifyPropertyChanged*, bound controls receive notifications of property changes when items are updated with the following code.

### C#

```
m_showing.Name = NameTextBox.Text;  
m_showing.Website = WebsiteTextBox.Text;
```

### Visual Basic

```
m_showing.Name = NameTextBox.Text  
m_showing.Website = WebsiteTextBox.Text
```

You can use the Add panel to add new items to the combo box. You add items to the combo box by clicking the Add Item button. The combo box is automatically updated to reflect the newly added item. The code for adding an item is as follows.

### C#

```
Company company = new Company();  
  
company.Id = int.Parse(AddIdTextBox.Text);  
company.Name = AddNameTextBox.Text;  
company.Website = AddWebsiteTextBox.Text;  
  
m_datasource.Add(company);
```

### Visual Basic

```
Dim company As Company = New Company()  
  
company.Id = Integer.Parse(AddIdTextBox.Text)  
company.Name = AddNameTextBox.Text  
company.Website = AddWebsiteTextBox.Text  
  
m_datasource.Add(company)
```

You remove the selected item in the combo box by clicking the Remove Item button. The following shows the code for removing an item.

**C#**

```
if (comboBox1.SelectedIndex >= 0)
{
    m_datasource.RemoveAt(comboBox1.SelectedIndex);
}
```

**Visual Basic**

```
If (comboBox1.SelectedIndex >= 0) Then
    m_datasource.RemoveAt(comboBox1.SelectedIndex)
End If
```

## Binding with the *ListBox* Control

The *ListBoxBinding* window demonstrates how to use the binding classes with a list box, which is created using the *ListBox* class in WPF and Silverlight. An object that implements *IEnumerable* can bind to a list box by using the following code.

**C#**

```
ExampleListBox.ItemsSource = m_datasource;
ExampleListBox.DisplayMemberPath = "Name";
ExampleListBox.SelectedValuePath = "Id";
```

**Visual Basic**

```
ExampleListBox.ItemsSource = m_datasource
ExampleListBox.DisplayMemberPath = "Name"
ExampleListBox.SelectedValuePath = "Id"
```

However, the list box does not receive dynamic updates unless the collection implements *INotifyCollectionChanged*. To have the list box receive notifications, you use the *ObservableCollection(T)* class, which implements *INotifyCollectionChanged* as well as the *NotificationList(T)* you created earlier in this chapter.

The Update panel shows the currently selected item in the list box. You update the selected item by clicking the Update Item button. The list box is automatically updated to reflect the updated item. Because each item implements *INotifyPropertyChanged*, bound controls receive notifications of property changes when items are updated with the following code.

**C#**

```
m_showing.Name = NameTextBox.Text;
m_showing.Website = WebsiteTextBox.Text;
```

**Visual Basic**

```
m_showing.Name = NameTextBox.Text
m_showing.Website = WebsiteTextBox.Text
```

You can use the Add panel to add new items to the list box. You add items to the list box by clicking the Add Item button. The list box is automatically updated to reflect the newly added item. The code for adding an item is as follows.

**C#**

```
Company company = new Company();

company.Id = int.Parse(AddIdTextBox.Text);
company.Name = AddNameTextBox.Text;
company.Website = AddWebsiteTextBox.Text;

m_datasource.Add(company);
```

**Visual Basic**

```
Dim company As Company = New Company()

company.Id = Integer.Parse(AddIdTextBox.Text)
company.Name = AddNameTextBox.Text
company.Website = AddWebsiteTextBox.Text

m_datasource.Add(company)
```

You remove the selected item in the list box by clicking the Remove Item button. The following shows the code for removing an item.

**C#**

```
if (comboBox1.SelectedIndex >= 0)
{
    m_datasource.RemoveAt(listBox1.SelectedIndex);
}
```

**Visual Basic**

```
If (comboBox1.SelectedIndex >= 0) Then
    m_datasource.RemoveAt(listBox1.SelectedIndex)
End If
```

## Binding with the *ListView* Control

The *ListViewBinding* window demonstrates how to use the binding classes with a list view, which is created using the *ListView* class in WPF and Silverlight. An object that implements *IEnumerable* can bind to a list view by using the following code.

**C#**

```
ExampleListView.ItemsSource = m_datasource;
ExampleListView.DisplayMemberPath = "Name";
ExampleListView.SelectedValuePath = "Id";
```

**Visual Basic**

```
ExampleListView.ItemsSource = m_datasource  
ExampleListView.DisplayMemberPath = "Name"  
ExampleListView.SelectedValuePath = "Id"
```

However, the list view does not receive dynamic updates unless the collection implements *INotifyCollectionChanged*. To have the list view receive notifications, you use the *ObservableCollection(T)* class, which implements *INotifyCollectionChanged* as well as the *NotificationList(T)* you created earlier in this chapter.

The Update panel shows the currently selected item in the list view. You update the selected item by clicking the Update Item button. The list view is automatically updated to reflect the updated item. Because each item implements *INotifyPropertyChanged*, bound controls receive notifications of property changes when items are updated with the following code.

**C#**

```
m_showing.Name = NameTextBox.Text;  
m_showing.Website = WebsiteTextBox.Text;
```

**Visual Basic**

```
m_showing.Name = NameTextBox.Text  
m_showing.Website = WebsiteTextBox.Text
```

You can use the Add panel to add new items to the list view. You add items to the list view by clicking the Add Item button. The list view is automatically updated to reflect the newly added item. The code for adding an item is as follows.

**C#**

```
Company company = new Company();  
  
company.Id = int.Parse(AddIdTextBox.Text);  
company.Name = AddNameTextBox.Text;  
company.Website = AddWebsiteTextBox.Text;  
  
m_datasource.Add(company);
```

**Visual Basic**

```
Dim company As Company = New Company()  
  
company.Id = Integer.Parse(AddIdTextBox.Text)  
company.Name = AddNameTextBox.Text  
company.Website = AddWebsiteTextBox.Text  
  
m_datasource.Add(company)
```

You remove the selected item in the list view by clicking the Remove Item button. The following shows the code for removing an item.

**C#**

```
if (comboBox1.SelectedIndex >= 0)
{
    m_datasource.RemoveAt(listBox1.SelectedIndex);
}
```

**Visual Basic**

```
If (comboBox1.SelectedIndex >= 0) Then
    m_datasource.RemoveAt(listBox1.SelectedIndex)
End If
```

## Binding with the *TreeView* Control

The *TreeViewBinding* window demonstrates how to use the binding classes with a tree view, which is created using the *TreeView* class in WPF and Silverlight. An object that implements *IEnumerable* can bind to a tree view by using the following code.

**C#**

```
ExampleTreeBox.ItemsSource = m_datasource;
ExampleTreeBox.DisplayMemberPath = "Name";
ExampleTreeBox.SelectedValuePath = "Id";
```

**Visual Basic**

```
ExampleTreeBox.ItemsSource = m_datasource
ExampleTreeBox.DisplayMemberPath = "Name"
ExampleTreeBox.SelectedValuePath = "Id"
```

However, the tree view does not receive dynamic updates unless the collection implements *INotifyCollectionChanged*. To have the tree view receive notifications, you use the *ObservableCollection(T)* class, which implements *INotifyCollectionChanged* as well as the *NotificationList(T)* you created earlier in this chapter.

The tree view control is capable of showing hierarchical data. Each node in the tree view can be displayed using other controls, such as a *TextBlock*, *Button*, and so on. The children of the nodes and the controls used to display them can be defined using the *HierarchicalDataTemplate* class. The *TreeViewBinding* window uses a custom node class called *TreeNode* for each of the nodes. Each child node is exposed through the *Nodes* property of *TreeNode* as a *NotificationList(TreeNode)*. The *NotificationList(T)* class is used instead of *ObservableCollection(T)*. The *NotificationList(T)* class lets you receive the items that were cleared when the *Clear* method is called. The *Text* property of the *TreeNode* stores the text of each node. The following code shows how the tree view is mapped to the custom *TreeNode* class.

**XAML**

```
<TreeView x:Name="ExampleTreeView">
    <TreeView.Resources>
        <HierarchicalDataTemplate DataType="{x:Type src:TreeNode}"
            ItemsSource="{Binding Path=Nodes}">
            <TextBlock Text="{Binding Path=Text}" />
        </HierarchicalDataTemplate>
    </TreeView.Resources>
</TreeView>
```



**More Info** You can find more information about using the *HierarchicalDataTemplate* on MSDN at <http://msdn.microsoft.com/en-us/library/system.windows.hierarchicaldatatemplate.aspx>.

With the Update button, you can change the text of the currently selected node with the contents in the Text text box. The tree view is automatically updated to reflect the updated item. Because each item implements *INotifyPropertyChanged*, bound controls receive notifications of property changes when items are updated with the following code.

**C#**

```
TreeNode node = (TreeNode)ExampleTreeView.SelectedItem;
node.Text = TextTextBox.Text;
```

**Visual Basic**

```
Dim node as TreeNode = CType(ExampleTreeView.SelectedItem, TreeNode)
node.Text = TextTextBox.Text
```

You can use the Add button to add a new child node to the currently selected node. The tree view is then automatically updated to reflect the newly added item. The code for adding an item is as follows.

**C#**

```
TreeNode node = (TreeNode)ExampleTreeView.SelectedItem;

if (node == null)
{
    m_nodes.Add(new TreeNode(TextTextBox.Text));
}
else
{
    node.Nodes.Add(new TreeNode(TextTextBox.Text));
}
```

**Visual Basic**

```
Dim node as TreeNode = CType(ExampleTreeView.SelectedItem, TreeNode)

If (node is Nothing) Then
    m_nodes.Add(new TreeNode(TextTextBox.Text))
Else
    node.Nodes.Add(new TreeNode(TextTextBox.Text))
End If
```

The newly created node is added to the root if no nodes are currently selected.

You remove the selected item in the *TreeView* by clicking the Remove button. The following shows the code for removing an item.

#### C#

```
TreeViewNode node = (TreeViewNode)ExampleTreeView.SelectedItem;

if (node.Parent != null)
{
    node.Parent.Nodes.Remove(node);
}
else
{
    m_nodes.Remove(node);
}
```

#### Visual Basic

```
Dim node As TreeViewNode = CType(ExampleTreeView.SelectedItem, TreeViewNode)

If (node.Parent IsNot Nothing) Then
    node.Parent.Nodes.Remove(node)
Else
    m_nodes.Remove(node)
End If
```

## Binding with the *CollectionView* Class

The *CollectionViewBinding* window demonstrates how to use the binding classes with the *BindingListCollectionView* and *ListCollectionView* classes. Binding to the list is done as follows.

#### C#

```
ExampleListView1.ItemsSource = new ListCollectionView(DL.GetDataSource());
ExampleListView2.ItemsSource = new ListCollectionView(DL.GetDataSource());
ExampleListView3.ItemsSource = new BindingListCollectionView(DL.GetDataSourceBL());
```

#### Visual Basic

```
ExampleListView1.ItemsSource = new ListCollectionView(DL.GetDataSource())
ExampleListView2.ItemsSource = new ListCollectionView(DL.GetDataSource())
ExampleListView3.ItemsSource = new BindingListCollectionView(DL.GetDataSourceBL())
```

Once bound, *ListCollectionView* can be filtered using the filtering controls present on the window. The following code shows how the views are filtered when the user clicks the filter button.

C#

```
private void FilterButton_Click(object sender, RoutedEventArgs e)
{
    ICollectionView view = null;

    ComboBoxItem cbi = (ComboBoxItem)FilterView.SelectedItem;

    switch (Convert.ToInt32(cbi.Content))
    {
        case 1:
            view = (ICollectionView)ExampleListView1.ItemsSource;
            break;
        case 2:
            view = (ICollectionView)ExampleListView2.ItemsSource;
            break;
    }

    if (!view.CanFilter)
    {
        return;
    }

    view.Filter = null;

    PropertyDescriptor pd = (PropertyDescriptor)FilterProperty.SelectedItem;
    string text = FilterText.Text;

    cbi = (ComboBoxItem)FilterOperator.SelectedItem;
    switch (cbi.Content.ToString())
    {
        case ">":
            view.Filter = item => { return Comparer.Default.Compare(pd.GetValue(item),
                pd.Converter.ConvertFromString(text)) > 0 ; };
            break;
        case ">=":
            view.Filter = item => { return Comparer.Default.Compare(pd.GetValue(item),
                pd.Converter.ConvertFromString(text)) >= 0; };
            break;
        case "=":
            view.Filter = item => { return Comparer.Default.Compare(pd.GetValue(item),
                pd.Converter.ConvertFromString(text)) == 0; };
            break;
        case "<":
            view.Filter = item => { return Comparer.Default.Compare(pd.GetValue(item),
                pd.Converter.ConvertFromString(text)) < 0; };
            break;
        case "<=":
            view.Filter = item => { return Comparer.Default.Compare(pd.GetValue(item),
                pd.Converter.ConvertFromString(text)) <= 0; };
            break;
    }
}
```

### Visual Basic

```
Private Sub FilterButton_Click(ByVal sender As System.Object, _
                               ByVal e As System.Windows.RoutedEventArgs)
    Dim view As ICollectionView = Nothing

    Dim cbi As ComboBoxItem = CType(FilterView.SelectedItem, ComboBoxItem)

    Select Case (Convert.ToInt32(cbi.Content))
        Case 1
            view = CType(ExampleListView1.ItemsSource, ICollectionView)
        Case 2
            view = CType(ExampleListView2.ItemsSource, ICollectionView)
    End Select

    If (Not view.CanFilter) Then
        Return
    End If

    view.Filter = Nothing

    Dim pd As PropertyDescriptor = CType(FilterProperty.SelectedItem, PropertyDescriptor)
    Dim text As String = FilterText.Text

    cbi = CType(FilterOperator.SelectedItem, ComboBoxItem)
    Select Case (cbi.Content.ToString())
        Case ">"
            view.Filter = Function(item) Comparer.Default.Compare(pd.GetValue(item), _
                                                    pd.Converter.ConvertFromString(text)) > 0
        Case ">="
            view.Filter = Function(item) Comparer.Default.Compare(pd.GetValue(item), _
                                                    pd.Converter.ConvertFromString(text)) >= 0
        Case "="
            view.Filter = Function(item) Comparer.Default.Compare(pd.GetValue(item), _
                                                    pd.Converter.ConvertFromString(text)) = 0
        Case "<"
            view.Filter = Function(item) Comparer.Default.Compare(pd.GetValue(item), _
                                                    pd.Converter.ConvertFromString(text)) < 0
        Case "<="
            view.Filter = Function(item) Comparer.Default.Compare(pd.GetValue(item), _
                                                    pd.Converter.ConvertFromString(text)) <= 0
    End Select
End Sub
```

The *ListCollectionView* and *BindingListCollectionView* can be sorted by clicking the headers on the list view. The following code shows how the views are sorted when the user clicks a header.

C#

```
void GridViewColumnHeaderClickedHandler(object sender, RoutedEventArgs e)
{
    GridViewColumnHeader headerClicked = e.OriginalSource as GridViewColumnHeader;
    ListView lv = (ListView)sender;
    ICollectionView view = (ICollectionView)lv.ItemsSource;

    if (!view.CanSort)
    {
        return;
    }

    if (headerClicked != null)
    {
        if (headerClicked.Role != GridViewColumnHeaderRole.Padding)
        {
            string header = headerClicked.Column.Header as string;

            if (view.SortDescriptions.Count > 0)
            {
                ListSortDirection direction = ListSortDirection.Ascending;

                foreach (var sd in view.SortDescriptions)
                {
                    if (sd.PropertyName == header)
                    {
                        if (direction == sd.Direction)
                        {
                            direction = ListSortDirection.Descending;
                        }
                        break;
                    }
                }

                view.SortDescriptions.Clear();
                view.SortDescriptions.Add(new SortDescription(header, direction));
            }
            else
            {
                view.SortDescriptions.Add(new SortDescription(header,
                                                               ListSortDirection.Ascending));
            }
        }
    }
}
```

### Visual Basic

```
Sub GridViewColumnHeaderClickedHandler(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim headerClicked As TryCast(e.OriginalSource, GridViewColumnHeader)
    Dim lv As ListView = CType(sender, ListView)
    Dim view As ICollectionView = CType(lv.ItemsSource, ICollectionView)

    If (Not view.CanSort) Then
        Return
    End If

    If (headerClicked IsNot Nothing) Then
        If (headerClicked.Role <> GridViewColumnHeaderRole.Padding) Then
            Dim header As String = CType(headerClicked.Column.Header, String)

            If (view.SortDescriptions.Count > 0) Then
                Dim direction As ListSortDirection = ListSortDirection.Ascending

                For Each sd As SortDescription In view.SortDescriptions
                    If (sd.PropertyName = header) Then
                        If (direction = sd.Direction) Then
                            direction = ListSortDirection.Descending
                        End If
                        Exit For
                    End If
                Next

                view.SortDescriptions.Clear()
                view.SortDescriptions.Add(New SortDescription(header, direction))
            Else
                view.SortDescriptions.Add(New SortDescription _
                    (header, ListSortDirection.Ascending))
            End If
        End If
    End If
End Sub
```

## Summary

In this chapter, you saw how to use collections with controls in WPF and Silverlight. The built-in notification interfaces for WPF and Silverlight can speed up productivity by implementing common operations that are done between collections and controls. These operations allow the controls to display collection changes and enable your collections to interact with the users.



# Index

## A

Action(T) method, 225–228  
AddAfter method  
  doubly linked lists, 60, 64  
  linked lists, 273, 274  
  singly linked lists, 34–35  
AddBefore method  
  doubly linked lists, 60, 64  
  linked lists, 273, 275  
  singly linked lists, 34–36  
AddFirst method (linked lists), 273, 275  
AddIndex method, 559  
AddLast method (linked lists), 273, 276  
Add method  
  arrays and, 10  
  association lists, 92  
  calling in BindingSource, 575  
  List(T) class, 223–224  
  to associate items with keys, 297  
  for users to associate values with keys, 123  
AddNew method, 541  
AddRange method  
  adding to Synchronized Wrapper class, 486  
  List(T) class, 223–224  
Add(TKey, TValue, bool) method, 93–94, 124, 126  
AddToBeginning method  
  doubly linked lists, 60  
  singly linked lists, 34  
AddToEnd method  
  doubly linked lists, 61  
  singly linked lists, 35  
AgeComparison method (sorting lists), 239  
algorithms, Quicksort, 232  
allOccurrences flag, 17  
AllowEdit property, 541  
AllowNew property, 541  
AllowRemove property, 541  
And method (BitArray class), 313–314  
AND operations, 431  
anonymous methods, defined, 219  
anonymous types, 452, 457–458  
appending items to List(T) class, 223–224  
ApplyFilter method, 566  
ApplySort method, 553, 568, 573–574  
ArrayDebugView class, 8  
ArrayEx(T) class  
  adding IList(T)/IList support to, 429–433  
  collection support for, 398–400  
  definition, changing, 521–522  
  enumeration support for, 349–354  
  generic class, 6

providing synchronization support to, 510  
serializing, 522–524  
using, 23–26  
arrays  
  advantages of, 4  
  allowing users to add items, 10–12  
  allowing users to remove items, 13–17  
  ArrayList class, 19  
  array index value, 20  
  ArrayList class, 4, 6  
  associative. *See associative arrays*  
  converting array elements to strings, 82–83  
  creating simple, 5–8  
  defined, 3  
  disadvantages of, 4  
  dynamic, 4, 222  
  elements in, 5  
  helper function to copy contents of, 22  
  helper methods and properties, adding, 17–22  
  implementing queues with, 154–155  
  indexing, 5  
  performing selection sorts on, 25  
  of references, 3  
  removing items from, 4  
  uses of, 3–4  
  using to create linked lists, 81  
  using to implement stacks. *See stacks*  
  of values, 3  
  writing sorted lists to console, 26  
Associate method, 111  
association lists  
  advantages/disadvantages of, 88  
  allowing users to associate values with keys, 92–94  
  allowing users to remove keys, 95–98  
  AssociativeArrayAL(T) class, creating, 89–92  
  helping methods and properties, adding, 98–104  
AssociativeArrayAL(TKey, TValue) class  
  adding collection support to, 417–422  
  adding Key/Value pair support to, 436–440  
  enumeration support for, 385–391  
AssociativeArrayHT(T) class  
  adding fields to, 118  
  constructors for, 122–123  
  creating, 119–122  
AssociativeArrayHT(TKey, TValue) class  
  adding collection support to, 422–427  
  adding Key/Value pair support to, 437–440  
  enumeration support for, 391–396  
associative arrays  
  advantages/disadvantages of, 88  
  association lists for. *See association lists*  
  AssociativeArrayAL(T) class, creating, 89–92

associative arrays (*continued*)  
 Associative Array class, using, 147–152  
 defined, 87  
 serializing, 528–531  
 synchronization support for, 512  
 use of, 87  
 using hash tables for. *See hash tables for associative arrays*  
 AutoResetEvent, 478  
 AverageArray method, 491, 494–495, 497  
 Average method (LINQ queries), 448

## B

BinarySearch methods, 222, 247, 259–263  
 binding  
 with BindingSource object, 581–582  
 BindingListCollectionView, 597–598, 605  
 BindingList(T) class, 574  
 BindingList(T) in .NET Framework, 541  
 BindingSource class, 575–576  
 with CollectionView class, 605–608  
 with ComboBox control, 576–577, 598–599  
 with DataGridView control and IBindingList, 578–580  
 with DataGridView control and IBindingListView, 580  
 with IBindingList, 578–580  
 with ListBox control, 577–578, 600–601  
 with ListView control, 601–602  
 with TreeView control, 603–605  
 BitArray class  
 accessing bits in, 308–310  
 And method, 313–314  
 BitArray And(BitArray value), 313–314  
 BitArray Not(), 313  
 BitArray Or(BitArray value), 314  
 creating BitArrays, 306–307  
 Not method, 313  
 Or method, 314–315  
 overview, 306  
 using, 306  
 using for bit operations, 310–313  
 Xor method, 314–315  
 bitwise equality, 216–218  
 bool Add(T item), 330  
 bool ContainsKey(TKey key), 301  
 bool Contains(T item), 265–266, 287–288, 291, 338  
 bool Contains(T value), 281  
 bool ContainsValue(TValue value), 302–303  
 bool Exists(Predicate(T) match), 266–267  
 bool Get(int index), 309  
 bool IsProperSubsetOf(IEnumerable(T) other), 333–334  
 bool IsSubsetOf(IEnumerable(T) other), 335  
 bool IsSupersetOf(IEnumerable(T) other), 336  
 bool Item[int index] {get; set; }, 309  
 bool Overlaps(IEnumerable(T) other), 336–337  
 bool Remove(LinkedListNode(T) node), 277–278  
 bool Remove(T item), 228, 331–332  
 bool Remove(TKey key), 298

bool Remove(T value), 277  
 bool TrueForAll(Predicate(T) match), 267–269  
 buckets  
 hash table, 106–107  
 picking number of, 113–116  
 sharing nodes across, 117–118

## C

cafeteria plates simulation (example)  
 creating variables and plate stack, 188  
 customer in line receiving plate, 191–193  
 eating on/carrying plates to be cleaned, 189–190  
 looping application pending key press, 188–189  
 Plate class, defining, 187–188  
 plates being cleaned, 190–191  
 CalculateCapacity method, 130  
 CalculateHashCode method, 136  
 CanCastToT method, 431, 433  
 CancelNew implementation, 543  
 capacity  
 Capacity property, 9, 17–22  
 of circular buffers, changing, 207–209  
 of lists, modifying, 228  
 setting to Count, 524  
 chaining (hash tables), 110–111  
 checking  
 dictionaries, 301–304  
 stacks, 291–292  
 circular buffers  
 adding helper methods/properties to CircularBuffer(T) class, 205–207  
 allowing users to add items to CircularBuffer(T) class, 201–203  
 allowing users to remove items from CircularBuffer(T) class, 203–204  
 changing capacity of, 207–209  
 CircularBuffer(T) class, using. *See song request line simulator example using CircularBuffer(T)*  
 constructors, creating in CircularBuffer(T) class, 198–200  
 defined, 193  
 implementing in CircularBuffer(T) class, 194–196  
 internal storage, 195–198  
 uses of, 193  
 CircularBuffer(T) class  
 adding collection support to, 401–403  
 enumeration support for, 355–360  
 C# language, This keyword in, 20  
 classes  
 adding collection support to. *See see collection support, adding to classes*  
 adding enumeration support to. *See enumeration support, adding to classes*  
 adding IList(T)/IList support to. *See IList(T)/IList support, adding to classes*  
 adding Key/Value pair support to, 435  
 cleaning stacks, 292

ClearedItems property, 319

Clear method

  associative arrays and, 131

  associative lists and, 95

  calling in BindingSource, 575

  CircularBuffer(T) class and, 204

  in doubly linked lists, 74

  HashSet(T) class and, 331

  linked lists and, 277

  QueuedLinkedList(T) class and, 167

  Queue(T) class and, 285

  to remove items from lists, 228

  to remove items from stacks, 290–291

  to remove keys and values from dictionaries, 298–299

  to remove nodes from lists, 48

  to remove items from ArrayEx(T) class, 13

CollectionBase class

  OnClear/OnClearComplete methods, 318–319

  OnInsert/OnInsertComplete methods, 319–323

  OnRemove/OnRemoveComplete methods, 321–322

  OnSet/OnSetComplete methods, 323–324

  OnValidate method, 317–318

  overview, 317

CollectionChanged events, 583, 595–596

collections

  collection classes, 480

  collection initialization, 453

  enumerating over while changing, 511

  serializing. *See* serializing collections

  synchronized collection classes, 511–512

  using in WPF and Silverlight, 583

  using threads with. *See* threads

  using with Windows Forms. *See* Windows Forms

collection support, adding to classes

  ArrayEx(T) class, 398–400

  AssociativeArrayAL(TKey,TValue) class, 417–422

  AssociativeArrayHT(TKey,TValue) class, 422–427

  CircularBuffer(T) class, 401–403

  DoubleLinkedList(T) class, 403–408

  overview, 398

  QueudLinkedList(T) class, 411–413

  QueuedArray(T) class, 408–410

  SingleLinkedList(T) class, 403–408

  StackedArray(T) class, 412–414

  StackedLinkedList(T) class, 415–418

CollectionView class, binding with, 605–608

collisions, hash table, 107–109

ComboBox control

  binding with, 576–577, 598–599

  setting DisplayMember property of, 539–540

CompareExchange method, 482

Compare function (sorting), 555

Compare method, 245

CompareTo(T) method, 218

Comparison(T) method (sorting), 236

constructors

  ArrayEx(T) class, 8–10

  AssociativeArrayAL(T) class, 91–92

AssociativeArrayHT(T) class, 122–123

BitArray(Boolean[] values), 307

BitArray(Byte[] bytes), 307–308

BitArray(Int32 length), 308

BitArray(Int32 length, Boolean defaultValue), 307

BitArray(Int32[] values), 308

CircularBuffer(T) class, 198–200

creating in QueuedArray(T) class, 155–157

creating in QueuedLinkedList(T) class, 164–165

Dictionary(TKey, TValue)(), 294

Dictionary(TKey, TValue)(IDictionary(TKey, TValue) dictionary), 294–296

Dictionary(TKey, TValue)(IEqualityComparer(TKey) comparer), 295–296

Dictionary(TKey, TValue)(int capacity), 296

Dictionary(TKey, TValue)(int capacity, IEqualityComparer(TKey) comparer), 296

DoubleLinkedList(T) class, 59–60

LinkedList(T), 273

LinkedList(T)(IEnumerable(T) collection), 273

List(T) class, 222

Queue(T)(), 284

Queue(T)(IEnumerable(T)), 284

Queue(T)(int size), 284

SingleLinkedList(T) class, 33–34

StackedArray(T) class, 178–179, 183–184

Stack(T), 288

Stack(T)(IEnumerable(T)), 288–289

Stack (T)(int size), 289

ContainsKey method

  for checking dictionaries, 301

  for discovering keys/values in collections, 101, 141

Contains method

  to allow users to view array contents, 17–22

  to check for items in lists, 50

  in checking list contents, 265–266

  default equality comparer and, 222

  to discover items in stacks, 291

  in doubly linked lists, 76–77

  HashSet(T) class, 338

  in queues, 159

  linked lists, 279

  QueuedLinkedList(T) class, 167

  Queue(T) class, 287–288

  to view contents of lists, 49–50

ContainsValue method

  to check for keys/values in collections, 101, 141

  for checking dictionary values, 302

contents of lists, checking, 263–269

ConvertAll method, 269

coprime multiplier, 113

copying

  contents of internal arrays, 53, 79

  CopyTo method, 400, 406–408

Count method

  LINQ queries and, 448

  setting Capacity to, 524

Count property  
 ArrayEx(T) classes, 17–22  
 association lists, 98  
 associative arrays, 136  
 doubly linked lists, 60  
 example, 49–50  
 for returning key value pairs in dictionaries, 303  
 for returning number of items in stacks, 292–293  
 linked lists, 279  
 passing to Array.Copy method, 400  
 QueuedLinkedList(T) class, 167  
 in queues, 159  
 for returning number of items in queues, 287  
 count variable (circular buffers), 197  
 CurrentReadCount property, 497

## D

data  
 breaking up with hash table buckets, 106–107  
 serializing/deserializing, 520–521  
 serializing to memory, 515–516  
 data binding  
 simple (Window Forms), 539–540  
 two-way. *See* two-way data binding  
 DataGridView, 541, 543  
 DataGridViewAdvanceBinding form, 580  
 DataGridView control  
 and IBindingList, binding with, 578–580  
 and IBindingListView, binding with, 580  
 data sources  
 DataSource property (ListBox control), 539  
 picking (from clause), 453–458  
 used with LINQ, 442–443  
 deadlocking, 485–486  
 DebuggerDisplayAttribute, 8  
 DebuggerTypeProxyAttribute, 8  
 default equality comparison, 215  
 DefaultIfEmpty method, 466  
 default ordering comparer, 218  
 default Person structure (searching lists), 250–251  
 delegates, defined, 219  
 Dequeue method, 158, 166, 286–287  
 Deserialize method, 516  
 DevGuideToCollections project, 6  
 DictionaryBase class  
 OnClear/OnClearComplete methods, 325  
 OnGet method, 325  
 OnInsert/OnInsertComplete methods, 325–326  
 OnRemove/OnRemoveComplete methods, 326–327  
 OnSet/OnSetComplete methods, 327–328  
 OnValidate method, 324–325  
 overview, 324  
 Dictionary property, 324  
 Dictionary(T) class, 512  
 Dictionary(TKey,TValue) class  
 adding items to dictionaries, 297–298  
 checking dictionaries, 301–304

creating dictionaries, 294–296  
 Dictionary< TKey,TValue>.KeyCollection Keys, 304–305  
 Dictionary< TKey,TValue>.ValueCollection Values,  
 305–306  
 overriding GetHashCode method in, 114  
 overview, 292  
 removing items from dictionaries, 298–299  
 retrieving values from dictionaries with keys, 299–301  
 using, 293–294  
 DisplayMember property (ComboBox control), 539  
 Dispose method  
 AssociativeArrayAL(TKey,TValue) class, 388  
 AssociativeArrayHT(TKey,TValue) class, 394  
 CircularBuffer(T) class, 357  
 defined in IDisposable interface, 352  
 DoubleLinkedList(T) class, 364  
 QueuedArray(T) class, 370  
 StackedArray(T) class, 377  
 StackedLinkedList(T) class, 382  
 Distinct method, 468  
 Division method, 112  
 DoubleLinkedList(T) class  
 collection support, adding to, 403–408  
 creating constructors in, 59–60  
 enumeration support for, 360–367  
 doubly linked lists  
 allowing users to add items, 60–68  
 allowing users to remove items, 68–75  
 constructors, creating, 59–60  
 DoubleLinkedListNode(T), 54  
 DoubleLinkedList(T) class, declaring, 57–59  
 helper methods and properties, 76–81  
 Node class, creating, 54–57  
 overview, 54  
 Driver console application, creating in Visual Studio, 23  
 dynamic arrays, 4, 222

## E

elements in arrays, 5  
 element.ToString() method, 456  
 empty associative arrays, 91, 122  
 empty buffers, 199  
 empty class, 122  
 empty lists, 118  
 empty List(T) instance, creating, 222–223  
 empty objects, 178  
 EndNew implementation, 543  
 Enqueue method, 157, 165, 285  
 Enter method, 479, 488–489  
 EnterReadLock method, 494–495  
 EnterWriteLock method, 497  
 Entry struct, 118  
 enumerating over collections, 511  
 enumeration support, adding to classes  
 ArrayEx(T) class, 349–354  
 AssociativeArrayAL(TKey,TValue) class, 385–391  
 AssociativeArrayHT(TKey,TValue) class, 391–396

CircularBuffer(*T*) class, 355–360  
 DoubleLinkedList(*T*) class, 360–367  
 overview, 349–350  
 QueuedArray(*T*) class, 367–372  
 QueuedLinkedList(*T*) class, 373–374  
 SingleLinkedList(*T*) class, 360–367  
 StackedArray(*T*) class, 374–378  
 StackedLinkedList(*T*) class, 379–384  
 enumerators, overview of, 345–348  
 equality comparer  
   bitwise equality basics, 216–218  
   in Contains method, 19  
   overview, 215  
   reference equality basics, 215–216  
 Equals(*T*) method, 215  
 ExceptWith method, 337–338  
 exclusive locks (thread synchronization), 478–479  
 Exist method, 266–267  
 Exit method, 479  
 ExitReadLock method, 494–495  
 ExitWriteLock method, 497

## F

FIFO (First In, First Out) collections, 153  
 filtering results (LINQ), 458–459  
 filtering support (*IBindingListView* interface), 563–568  
 FindAll method, 247  
 FindIndexData method, 560–562  
 FindIndex/FindLastIndex methods, 251–257  
 FindIndex method, 247  
 FindKey method, 102, 126, 146, 420  
 FindLast method, 279  
 Find method  
   doubly linked lists and, 76  
   linked lists and, 279  
   List(*T*) and, 247  
   to search for items with properties matching  
     keys, 557–558  
     to view contents of lists, 49–50  
 FindValue method, 102  
 First method (LINQ queries), 448  
 First property (linked lists), 279  
 flags, using bitwise operations on, 310–313  
 forcing immediate query executions (LINQ), 446–449  
 foreach statements  
   vs. from clauses, 454  
   interaction with two interfaces, 347–349  
   using with List(*T*) class, 224–225  
 from clauses (LINQ), 454–459  
 FullOperation property, 199

## G

GetEnumerator method  
 ArrayEx(*T*) class, 346–347, 350  
 AssociativeArrayHT(*TKey*,*TValue*) class, 392  
 DoubleLinkedList(*T*) class, 361

QueuedLinkedList(*T*) class, 374  
 StackedArray(*T*) class, 375  
 StackedLinkedList(*T*) class, 380  
 GetHashCode method, 112  
 Get method, accessing bits in arrays with, 309–310  
 GetObjectData method, 523  
 group clauses (LINQ), 461–463  
 "grow by" values, 9  
 GROW\_BY constant, 8  
 GROW\_BY size (arrays), 9

## H

hashing function, 105  
 HashSet(*T*) class  
   adding items to, 330–331  
   Add method, 330  
   Clear method, 331  
   creating, 329–330  
   HashSet(*T*) (*IEnumerable*(*T*)), 329  
   HashSet(*T*) (*IEnumerable*(*T*),  
     *IEqualityComparer*(*T*)), 330  
   HashSet(*T*) (*IEqualityComparer*(*T*)), 330–331  
   overview, 329  
   performing set operations on, 333–338  
   Remove method, 331–332  
   RemoveWhere method, 332–333  
   removing items from, 331–333  
 hash tables for associative arrays  
   advantages/disadvantages of, 105  
   allowing users to associate values with keys, 123–130  
   allowing users to remove keys, 130–134  
   AssociativeArrayHT(*T*) class, creating, 119–122  
   chaining, 110–111  
   constructors for AssociativeArrayHT(*T*) class, 122–123  
   defined, 105  
   hashing function, picking, 112  
   hash table buckets, 106–107  
   hash table collisions, 107–109  
   internal storage, implementing, 116–118  
   number of buckets, picking, 113–116  
   Object.GetHashCode, 112  
 Head method  
   doubly linked lists and, 78  
   singly linked lists and, 52  
 Head pointer, 48  
 Head property  
   doubly linked lists and, 60  
   nodes and, 34, 42  
   to view status of lists, 49–50  
 helper function to copy contents of arrays, 22  
 helper methods  
   adding to arrays, 17–22  
   adding to association lists, 98–104  
   adding to associative arrays, 136–145  
   adding to CircularBuffer(*T*) class, 205–207  
   adding to doubly linked lists, 76–81  
   adding to QueuedArray(*T*) class, 159–163

- helper methods (*continued*)
- adding to `QueuedLinkedList(T)` class, 167–169
  - adding to singly linked lists, 49–54
  - adding to `StackedArray(T)` class, 180–182, 185–187
  - to convert elements of arrays to strings, 24–25
- `HierarchicalDataTemplate`, 603
- 
- I
- `IBindingList` interface
- binding with `DataGridView` control, 578–580
  - list manipulation support, adding, 541–543
  - notification support, adding, 543–552
  - overview, 541
  - searching support, adding, 557–562
  - sorting support, adding, 553–557
- `IBindingListView` interface (Windows Forms)
- advanced sorting, adding, 568–574
  - binding objects with `DataGridView` control, 580
  - filtering support, adding, 563–568
- `ICancelAddNew` interface, 542
- `ICollection/ICollection(T)` interfaces, 397–398
- `ICollection.SyncRoot` property, 489–490
- `ICollectionView` interface
- `BindingListCollectionView`, when to use, 597–598
  - binding with `CollectionView` class, 605–608
  - binding with `ComboBox` control, 598–599
  - binding with `ListBox` control, 600–601
  - binding with `ListView` control, 601–602
  - binding with `TreeView` control, 603–605
  - overview, 596, 597
- `IComparer(T)` interface, 219
- `IDictionary/IDictionary(TKey,TValue)` interfaces, 434–435
- `IDisposable` interface, 352
- `IEnumerable/IEnumerator` interfaces, 345–348, 454
- `IEnumerable(T)` interface, 223, 350
- `IEquatable(T)` interface, 215, 218
- `IFormatter` interface, 516
- `IList/IList(T)` interfaces, 428–429
- `IList(T)/IList` support, adding to classes
- `ArrayEx(T)` class, 429–433
- illustrations
- allowing users to add items to collection, 34
  - allowing users to add items to doubly linked lists, 60–61
  - allowing users to remove items from collections, 42
  - arrays, indexing, 5
  - circular buffers, changing capacity of, 207
  - doubly linked lists, 54
  - hash table buckets, 106, 108
  - internal storage (circular buffers), 195–196
  - nodes in doubly linked lists, 69
  - sequence of nodes in doubly linked lists, 54
  - singly linked lists, 28
- incrementing `m_count` (arrays), 11
- indexing arrays, 5
- indexing collections, 557
- `IndexOf` method, 17–22, 222, 263–265
- `Index` property, 309
- initializing
- collections, 453
  - `Initialize` method, 8–10, 198
  - objects during creation, 452–453
- `InnerHashtable` property, 324
- inner joins, 463–465
- `INotifyCollectionChanged` interface
- implementing, 583–589
  - notifying users of cleared items, 590–592
  - `OnClear/OnClearComplete` methods, 590–592
  - overview, 583
- `INotifyPropertyChanged/INotifyPropertyChanging` interfaces, 541, 550
- `INotifyPropertyChanged` method, 576–577, 581
- inserting items into `List(T)` class, 230–231
- `Insert` method (arrays), 11
- `Insert` method, 230–231, 575
- `InsertRange` method, 230–231
- installing code samples, xvii
- `int Count`, 287, 292–293, 303
- `int Count { get; }`, 280–281
- integers
- creating random array of, 25, 81–85
  - `IntegerType` enumeration, 464–465
- `Interlocked.CompareExchange` method, 482
- interlocked operations (thread synchronization), 475
- `InternalObject` collection class (example), 487
- internal storage
- circular buffers and, 195–198
  - implementing (hash tables), 116–118
- `IntersectWith` method, 333
- `int FindIndex(Predicate(T) match)` method, 252
- `int FindLastIndex(Predicate(T) match)` method, 252
- `int IndexOf(T item, int index, int count)/int`
- `LastIndexOf(T item, int index, int count)` methods, 265–266
- `int IndexOf(T item, int index)/int LastIndexOf(T item, int index)` methods, 264
- `int IndexOf(T item)/int LastIndexOf(T item) methods`, 263–264
- `int RemoveAll(Predicate(T) match)` method, 229
- `InvalidOperationException`, 126, 275, 277
- `IQueryable(T)` interface, 454
- `IsEmpty` property
- association lists and, 99
  - checking for empty collections with, 137
  - looking at list status with, 49–50
  - `QueuedLinkedList(T)` class, 167
  - in queues, 159
  - searching lists and, 250
- `ISerializable` interface, 518–519
- `IsEven` method, 258, 267
- `IsFiltered` method, 566
- `IsFixedSize` property, 429
- `IsProperSubsetOf` method (`HashSet(T)` class), 333–334
- `IsProperSupersetOf` method, 334–335

IsReadOnly property, 405, 418  
 IsSorted property, 553  
 IsSubsetOf method, 335  
 IsSupersetOf method, 336  
 IsSynchronization method, 485  
 IsSynchronized method, 399  
 IsSynchronized property, 481–483  
 Item get property, 300–301  
 Item property  
   associating keys to values with, 123, 297  
   for retrieving values with keys, 103  
   to set contents at specific index, 20  
   using FindKey method with, 146  
 items  
   adding to arrays, 10–12  
   adding to dictionaries, 297–298  
   adding to HashSet(T) class, 330–331  
   adding to stacks, 289  
   allowing to remove items from StackedArray(T) class, 185  
   allowing users to add to CircularBuffer(T) class, 201–203  
   allowing users to add to collections, 34–42  
   allowing users to add to doubly linked lists, 60–68  
   allowing users to add to QueuedLinkedList(T) class, 165  
   allowing users to add to queues, 157–158  
   allowing users to add to StackedArray(T) class, 179, 184  
   allowing users to remove from arrays, 13–17  
   allowing users to remove from CircularBuffer(T) class, 203–204  
   allowing users to remove from collections, 42–49  
   allowing users to remove from doubly linked lists, 68–75  
   allowing users to remove from QueuedLinkedList(T) class, 166–167  
   allowing users to remove from queues, 158–159  
   allowing users to remove from StackedArray(T) class, 180  
   inserting into List(T) class, 230–231  
   removing from dictionaries, 298–299  
   removing from HashSet(T) class, 331–333  
   removing from List(T) class, 228–230  
   removing from queues (Queue(T) class), 285–286  
   removing from stacks, 290–291  
 ITypeList interface, 598

**J**

join clauses (LINQ), 463–466

**K****keys**

  allowing users to associate values with, 92–94, 123–130

allowing users to remove, 95–98, 130–134  
 KeyCollection class, 437  
 property, 99, 137, 304–305  
 retrieving values from dictionaries with, 299–301  
 Key/Value pair support, adding to classes  
   AssociativeArrayAL(TKey,TValue) class., 436–440  
   AssociativeArrayHT(TKey,TValue) class, 437–440  
   overview, 435  
 KeyValuePair(TKey,TValue) struct, 419  
 KVPair struct, 91

**L**

lambda expressions  
   to See Whether All Match, 268–269  
   using for Action(T) method, 227–228  
   using for Comparison(T) method, 238  
   using for FindAll method, 258  
   using for find operations, 249–250  
   using for matching (searching lists), 255, 257  
   using for sorting, 555  
   using to Check for Existence, 267–268  
   using to convert phone number, 271  
   using with RemoveAll method, 229  
   using with RemoveWhere, 332–333  
   in Visual Basic, 221  
   writing inline methods with, 220  
 LastIndexOf method  
   default equality comparer and, 222  
   searching lists with, 263–265  
 Last property (linked lists), 279  
 let clauses (LINQ), 466–468  
 LIFO (Last In, First Out)  
   collections, 175  
   stacks implemented as, 413  
 linked lists  
   advantages of, 27  
   classes, serializing, 524–528  
   class, using, 81–85  
   defined, 27  
   disadvantages of, 27  
   doubly linked lists. *See* doubly linked lists  
   LinkedListNode(T) AddAfter(LinkedListNode(T) node, T value), 274  
   LinkedListNode(T) AddBefore(LinkedListNode(T) node, T value), 275  
   LinkedListNode(T) AddFirst(T item), 275  
   LinkedListNode(T) AddLast (T item), 276  
   LinkedListNode(T) Find(T value)/LinkedListNode(T) FindLast(T value), 279–280  
   LinkedListNode(T) First { get; }/LinkedListNode(T) Last { get; }, 280  
   singly linked lists. *See* singly linked lists  
   uses of, 27–28  
   using arrays to create, 81  
   using to implement queues. *See* queues  
   using to implement stacks. *See* stacks

LinkedList(T) class

- adding nodes to linked lists, 273–276
- creating new linked lists, 273
- obtaining information about linked lists, 279–281
- overview, 272
- removing nodes from, 277–280
- using, 273

LINQ (Language Integrated Query)

- additions to .NET language for, 451–453
- basic query actions, 442
- data sources used with, 442–443
- filtering results (where clause), 458–459
- group clause, 461–463
- join clause, 463–466
- let clause, 466–468
- nesting statements with, 442
- ordering results (orderby clause), 460–461
- picking data source and selecting results, 453–458
- query execution with, 443–449
- querying data with and without (examples), 441–442
- structure for (examples), 449–450

ListBox control

- binding with, 577–578, 600–601
- setting DisplayMember property of, 539–540

ListChanged event, 543, 552, 584

ListChangeType.ItemChange notification, 552

ListCollectionView, 598, 605

lists

- association lists. *See association lists*
- doubly linked. *See doubly linked lists*
- list manipulation support (*IBindingList* interface), 541–543
- singly linked. *See singly linked lists*

List(T) classes

- appending items to, 223–224
- checking list contents, 263–269
- creating, 222–223
- inserting items into, 230–231
- List(T)(*IEnumerable*(T) collection), 223
- List(T)(int size), 223
- ListToString method, 232
- modifying lists, 269–272
- overview, 222
- removing items from, 228–230
- searching. *See searching List(T) classes*
- sorting. *See sorting List(T) classes*
- traversing, 224–228

List(T) FindAll(*Predicate*(T) match) method, 257–258

List(TOutput) ConvertAll(*Converter*(T, TOutput) converter), 269–271

ListView control, binding with, 601–602

locking mechanisms (threads), 474–475, 478–480, 487

LockRecursionPolicy.NoRecursion, 500

LockRecursionPolicy.SupportRecursion, 500

locks, upgradeable, 500–503

**M**

Main method (arrays), 26

mapping values to keys. *See hash tables for associative arrays*

m\_array field, 352, 376

Max method (LINQ queries), 448

m\_buffer field

- CircularBuffer(T) class, 357
- QueuedArray(T) class, 369

m\_capacity field, 118

m\_comparer field (association lists), 90

m\_count field, 8, 32, 59, 118

m\_current field, 363

m\_currentNode field, 387

m\_data field

- DoubleLinkedList(T) class, 183
- internal data storage with (queues), 155
- Node class, 57
- QueuedLinkedList(T) class, 164
- sorting, 554
- StackedArray(T) class, 178

m\_end field, 363

methods

- adding helper methods to doubly linked lists, 76–81
- anonymous, 219
- helper methods, adding to arrays, 17–22
- helper methods, adding to singly linked lists, 49–54
- to reverse order of array contents, 80

m\_head field, 32, 59

Microsoft .NET Framework 32-bit/64-bit applications, 3

Microsoft Visual Basic project, 6

Microsoft Visual C# class library project, 6

m\_index field, 352, 357

m\_list field, 363, 381

m\_newIndex field, 542, 543

m\_next field, 31, 57

modifying lists, 269–272

modulus (mod) operation, 196, 267

modulus operator, 112, 403

Monitor class, 478, 487–490

MoveNext method

- ArrayEx(T) class, 348, 353–354
- AssociativeArrayAL(TKey,TValue) class, 389, 390
- AssociativeArrayHT(TKey,TValue) class, 395
- CircularBuffer(T) class, 359
- DoubleLinkedList(T) class, 365–366
- QueuedArray(T) class, 371
- StackedArray(T) class, 378
- StackedLinkedList(T) class, 383
- traversing collections with, 139

m\_owner field, 31, 57

m\_prev field, 57

m\_syncRoot field, marking as nonserializable, 528, 529

m\_tail field

- doubly linked lists, 64
- DoublyLinkedList(T) class, 59
- to eliminate traversing complete lists, 37
- maintaining node references with, 32

multiline lambda statements, 221  
**m\_updateCode** field  
 association lists and, 90  
*AssociativeArrayHT(TKey,TValue)* class, 393  
*CircularBuffer(T)* class, 357  
 to determine changes in collections, 352  
*DoubleLinkedList(T)* class and, 59, 363  
*QueuedArray(T)* class, 369  
*QueuedLinkedList(T)* class, 164  
 queues, implementing with arrays, 155  
*StackedArray(T)* class, 178, 183  
*StackedLinkedList(T)* class, 381  
 storing value of before enumeration, 511  
 user list modification and, 8  
*Mutex* class, 478  
*MyStringComparer.GetHashCode* method, 116

## N

*NameToIndex* method, 107, 109  
**N** elements in arrays, 5  
 nested locks, 485  
.NET collections, using as LINQ data source, 443  
.NET Framework  
*ArrayList* class in, 6  
 hash codes and different versions of, 112  
 tools for thread synchronization, 475–479  
.NET language, LINQ additions to, 451–453  
**Next** field, 118  
**Next** method, 181  
**NextOperation** field, 188, 190  
 next pointers (linked lists), 27  
**Next** property  
 doubly linked lists, 60–61  
 nodes and, 34, 42  
**Node** class, creating  
 in doubly linked lists, 54–57  
 in singly linked lists, 28–30  
**nodes**  
 in doubly linked lists, 54, 68–69  
 removing, 48  
 removing from *LinkedList(T)* class, 277–280  
 non-exclusive locks (thread synchronization), 480–481  
**NonSerializable** attribute, 513, 516, 521  
*NotificationList(T)* class, 584  
 notification support (*IBindingList* interface), 543–552  
**Not** method (*BitArray* class), 313  
**NULL\_REFERENCE** constant, 118  
 null value, 50, 77

## O

*Object.Equals(Object)* method, 215, 218  
*Object.GetHashCode* method, 112  
**objects**  
 filtering list of, 459  
 initializing during creation, 452–453

*ObservableCollection(T)* class  
 overview, 593  
 recursive collection change events, 595–596  
 using, 593–594  
**oddInts** variable (LINQ), 448  
**OnClear/OnClearComplete** methods  
*CollectionBase* class, 318–319  
*DictionaryBase* class, 325  
**OnGet** method (*DictionaryBase* class), 325  
**OnInsert/OnInsertComplete** methods  
*CollectionBase* class, 319–323  
*DictionaryBase* class, 325–326  
 notification support and, 543  
**OnListChanged** method, 552  
**O(n)** operations, 222  
**OnRemove/OnRemoveComplete** methods  
*CollectionBase* class, 321–322  
*DictionaryBase* class, 326–327  
 notification support and, 543  
**OnSet/OnSetComplete** methods  
*CollectionBase* class, 323–324  
*DictionaryBase* class, 327–328  
**OnValidate** method  
*CollectionBase* class, 317–318  
*DictionaryBase* class, 324–325  
**OnXXX/OnXXXComplete** methods, 543, 584  
 operators, modulus, 112  
**orderby** clauses (LINQ), 460–461  
 ordering comparer, 218–219  
**Or** method (*BitArray* class), 314–315  
 outer joins, 465  
**Overlaps** method (*HashSet(T)* class), 336–337

## P

**Peek** method  
*QueuedLinkedList(T)* class, 167  
*Queue(T)* class, 286  
*StackedArray(T)* class, 181, 185–186  
 to view contents of queues, 159  
 to view items at top of stacks, 291  
 performance penalties, 9  
**PersonAgeComparer** class (list sorting), 241  
**PersonComparer** class (searching lists), 259–260  
**PersonNameComparer** class (sorting lists), 244  
 Person struct, 236  
 Person structure defined in *Sort(Comparison(T))*  
 method, 240  
**Plate** class, defining (stacks example), 187–188  
**Pop** method  
*QueuedLinkedList(T)* class and, 166  
 removing items from stacks with, 290–291  
 removing items from queues with, 158  
*StackedArray(T)* class and, 185  
 post-fix Attribute, 513  
**Predicate(T)** method, 247, 251, 257  
 previous pointers (linked lists), 27  
**Prev** property (doubly linked lists), 61–62

prime numbers, 113–115, 130  
 PrivateField field, 515  
 ProcessTask as Action(T) (example), 225–228  
 properties  
   adding to arrays, 17–22  
   adding to association lists, 98–104  
   adding to associative arrays, 136–145  
   adding to CircularBuffer(T) class, 205–207  
   adding to doubly linked lists, 76–81  
   adding to QueuedArray(T) class, 159–163  
   adding to QueuedLinkedList(T) class, 167–169  
   adding to singly linked lists, 49–54  
   property change notifications, 552  
 ProtectedField field, 515  
 PublicField field, 515  
 PublicObject collection class (example), 487  
 pushing items onto queues, 285  
 Push method  
   CircularBuffer(T) class, 202  
   QueuedArray(T) class, 157  
   QueuedLinkedList(T) class, 165  
   StackedArray(T) class, 179, 184  
   Stack(T) class, 289

## Q

query executions (LINQ), 443–449  
 Queue classes, serializing, 532–533  
 QueuedArray(T) class  
   adding collection support to, 408–410  
   enumeration support for, 367–372  
 QueuedLinkedList(T) class  
   adding collection support to, 411–413  
   enumeration support for, 373–374  
 queues  
   allowing users to add items to QueuedArray(T) class, 157–158  
   allowing users to add items to QueuedLinkedList(T) class, 165  
   allowing users to remove items from QueuedArray(T) class, 158–159  
   allowing users to remove items from QueuedLinkedList(T) class, 166–167  
   constructors, creating in QueuedArray(T) class, 155–157  
   constructors, creating in QueuedLinkedList(T) class, 164–165  
   defined, 153  
   helper methods and properties, adding to QueuedArray(T) class, 159–163  
   helper methods and properties, adding to QueuedLinkedList(T) class, 167–169  
   implementing with arrays, 154–155  
   implementing with linked lists, 162–163  
   QueuedArray(T) class, 154  
   QueuedLinkedList(T) class, 163

uses of, 153  
 using QueueArray(T)/ArrayEx(T) classes. *See song request line simulator using QueueArray(T) (example)*  
 Queue(T) class  
   adding to queues, 285–286  
   checking queues, 286–287  
   cleaning queues, 287  
   creating queues, 284  
   in Microsoft .NET Framework, 154–155  
   overview, 283  
   removing items from queues, 285–286  
 Quicksort algorithm (Array.Sort), 232–235

## R

Random class (random numbers), 25  
 Range class (enumeration example), 346–348  
 range variable elements, 455  
 ReaderWriterLock class, 480–481  
 ReaderWriterLockSlim class  
   for allowing access to multiple threads, 480–481  
   AverageArray method, 494–495  
   recursive locking and, 500  
   ReverseArray method, 490–493  
   State class, 493–495  
   SumArray method, 490–493  
   upgradeable locks, using, 500–503  
 ReapplyFilter method, 566  
 recursive collection change events, 595–596  
 recursive locking, 474–475, 500  
 reference equality, 215–218  
 references, arrays of, 3  
 RemoveAll method, 98, 134, 228  
 RemoveAt method, 14, 228, 575  
 Remove(DoubleLinkedListNode(KVPair)) method, 98  
 Remove(EntryData) method, 134, 136  
 RemoveFilterInternal function, 566  
 RemoveFilter method, 563, 568  
 RemoveFirst method, 277, 278  
 RemoveIndex method, 562  
 RemoveLast method, 277, 278–279  
 Remove method  
   arrays and, 10, 13–17  
   in associative arrays, 130  
   associative lists, 95  
   calling in BindingSource, 575  
   default equality comparer and, 222  
   in doubly linked lists, 74  
   HashSet(T) class, 331–332  
   linked lists, 277–278  
   nodes and, 42–49  
   to remove items from lists, 228  
   to remove values associated with specific keys, 298  
 RemoveRange method, 228  
 RemoveSort method, 556–557, 568–569

RemoveValue method  
 in associative arrays, 130  
 associative lists and, 95, 96  
 RemoveWhere method, 332–333  
 removing  
   items from dictionaries, 298–299  
   items from HashSet(T) class, 331–333  
   items from List(T) class, 228–230  
   items from queues (Queue(T) class), 285–286  
   items from stacks, 290–291  
   nodes from LinkedList(T) class, 277–280  
 Reset method  
   AssociativeArrayList(TKey,TValue) class, 391  
   AssociativeArrayList(TKey,TValue) class, 396  
 CircularBuffer(T) class, 360  
 DoubleLinkedList(T) class, 367  
 for COM interoperability, 346  
 QueuedArray(T) class, 372  
 StackedArray(T) class, 379  
 StackedLinkedList(T) class, 384  
 ReverseArray method, 470–473, 490–493  
 reversedNumbers variable, 334  
 Reverse method, 269, 272–273

## S

SaveUnsortedList method, 574  
 searching List(T) classes  
   BinarySearch methods, 259–263  
   FindIndex/FindLastIndex methods, 251–257  
   List(T) FindAll(Predicate(T) match) method, 257–258  
   overview, 247  
   T Find(Predicate(T) match)/T FindLast(Predicate(T)  
     match) methods, 247–252  
 searching support (IBindingList interface), 557–562  
 select clauses, 454–458  
 selection sorts, performing on arrays, 25  
 Semaphore class, 480  
 serializing collections  
   adding serialization support to collection  
     classes, 521–522  
   ArrayEx(T) Class, 522–524  
   ArrayEx(T) class definition, changing, 521–522  
   associative array classes, 528–531  
   controlling serialization behavior, 517–521  
   Deserialize method, 516  
   Formatter classes, 513–521  
   linked list classes, 524–528  
   Queue classes, serializing, 532–533  
   Serializable attribute, 513–517, 521  
   Serialize method, 516  
   Stack classes, serializing, 533–534  
 SetAll method, 309  
 Set method, 308  
 set operations, performing on HashSet(T) class,  
   333–338  
 set property (doubly linked lists), 57  
 signaling (thread synchronization), 476–479

Silverlight  
 ComboBox class and, 598  
 ListBox class and, 600  
 ListView class and, 601  
 TreeView class and, 603  
 using collections in, 583  
 simple data binding (Windows Forms), 539–540  
 SingleLinkedList(T) class  
   collection support, adding to, 403–408  
   enumeration support for, 360–367  
 singly linked lists  
   allowing users to add items, 34–42  
   allowing users to remove items, 42–49  
   constructors, creating, 33–34  
   helper methods and properties, adding, 49–54  
   Node class, creating, 28–30  
   overview, 28–29  
   SingleLinkedList(KVPair), 110–111  
   SingleLinkedListNode(T), defining, 28–30  
   SingleLinkedList(T) class, declaring, 31–33  
 song request line simulator using CircularBuffer(T)  
   (example), 210–211  
 song request line simulator using QueueArray(T)  
   (example)  
   drawing separator on screen, 171  
   loop pending keyboard press, 171–175  
   overview, 170–171  
   random collection of songs, creating, 171–172  
   variable declarations, 170  
 SortDescriptions property, 568  
 SortDirection property, 553, 570  
 sorted collections  
   overview, 339–340  
   SortedDictionary(TKey, TValue) class, 222, 339–341  
   SortedList(TKey, TValue) class, 339  
 sorted lists  
   SortedList(TKey, TValue) class, 222  
   writing to console (arrays), 26  
 sorting  
   advanced (IBindingListView interface), 568–574  
   BindingListCollectionView, 607  
   ListCollectionView, 607  
   sorting support (IBindingList interface), 553–557  
   Sort method, 222, 555  
   SortProperty property, 553, 570  
 sorting List(T) classes  
   built-in sorting, 232–235  
   overview, 231  
   void Sort(), 236  
   void Sort(Comparison(T) comparison), 236–239  
   void Sort(IComparer(T) comparer), 240–241  
   void Sort(int index, int count, IComparer(T)  
     comparer), 242–246  
 StackedArray(T) class  
   adding collection support to, 412–414  
   creating, 177–178  
   enumeration support for, 374–378

StackedLinkedList(T) class  
 adding collection support to, 415–418  
 enumeration support for, 379–384  
 stacks  
 adding common functionality, 175–176  
 allowing users to add items to StackedArray(T)  
     class, 179, 184  
 allowing users to remove items from StackedArray(T)  
     class, 180, 185  
 cafeteria plate simulation (example). *See* cafeteria  
     plate simulation (example)  
 constructors, creating in StackedArray(T) class,  
     178–179, 183–184  
 helper methods, adding to StackedArray(T)  
     class, 180–182, 185–187  
 serializing Stack classes, 533–534  
 StackedLinkedList(T) class, 182–183  
 uses of, 175  
 Stack(T) class  
     adding items to stacks, 289  
     checking stacks, 291–292  
     cleaning stacks, 292  
     creating stacks, 288–289  
     in .NET Framework, 175  
     overview, 288  
     removing items from stacks, 290–291  
 start/end pointers (circular buffers), 195, 207  
 State class, 493–495  
 storage, internal (circular buffers), 195–198  
 streams, serialization and, 516  
 strings  
     convert elements of arrays to, 24–25  
     converting array elements to, 82–83  
 structures as value types, 217  
 subexpressions, storing with let clauses, 466  
 SumArray method, 470–473, 490–493, 497  
 SupportsAdvancedSorting property, 568  
 SupportsChangeNotification flag, 552  
 SupportsChangeNotification property, 543  
 SupportsFiltering property, 563  
 SupportsSearching property, 557  
 SupportsSorting property, 553  
 synchronization, thread. *See* thread synchronization  
 synchronized collection classes, 511–512  
 SynchronizedCollection(T) class, 511  
 SynchronizedKeyedCollection(T) class, 512  
 SynchronizedReadOnlyCollection(T) class, 512  
 Synchronized Wrapper class, 504–510  
 SyncLock statement, 478  
 SyncRoot method, 399  
 SyncRoot property, 481–483, 483–486  
 System.Collections.Generic namespace, 215, 218  
 System.Collections.IEqualityComparer interface, 112  
 system requirements for exercises, xvi

**T**

Tail method, 52, 78  
 Tail property, 42, 49–50, 61  
 T Dequeue() method, 286–287  
 T Find(Predicate(T) match)/T FindLast(Predicate(T)  
     match) methods, 247–252  
 This keyword in C#, 20  
 threads  
     basics, 469  
     Monitor class, using, 487–490  
     ReaderWriterLockSlim class. *See* ReaderWriterLockSlim  
     class  
 thread synchronization  
     defined, 470  
     .NET Framework tools for, 475–479  
     supporting collection classes with, 480–482  
     Synchronized Wrapper class, implementing, 504–510  
     Synchronized Wrapper class vs. SyncRoot, 483–486  
     value of (example), 470–473  
     writing code as needed, 474–475  
 ToArray method (LINQ queries), 446  
 ToList method (LINQ queries), 446  
 ToString method, 8  
 T Peek() method, 286, 291–292  
 T Pop() method, 290  
 traversing List(T) class, 224–228  
 TreeView control, binding with, 603–605  
 TrimExcess method  
     to change capacity of stacks, 290, 292  
     HashSet(T) class, 338  
     to modify capacity of lists, 228  
     to reclaim memory (Queue(T) class), 287  
 Trim method, 269  
 TrueForAll method, 267–269  
 TryEnter method, 479, 488–489  
 TryEnterReadLock method, 496  
 TryGetValue method, 104, 146, 299–300  
 TValue Item[TKey key] { get; }, 300–301  
 TValue Item[TKey key] { set; }, 297  
 two-way data binding  
     IBindingList interface, implementing. *See* IBindingList  
     interface  
     overview, 540

**U**

UnionWith method (HashSet(T) class), 337  
 UnpackPhoneNumber method, 271  
 Update button (TreeView control), 604  
 Update Item button  
     binding with ListView control, 602  
     ComboBox control, 576  
 upgradeable locks, using, 500–503

users  
 allowing users to add items (`StackedArray(T)` class), 179  
 allowing users to add items to arrays, 10–12  
 allowing users to add items to `CircularBuffer(T)` class, 201–203  
 allowing users to add items to collections, 34–42  
 allowing users to add items to doubly linked lists, 60–68  
 allowing users to add items to `QueuedLinkedList(T)` class, 165  
 allowing users to add items to queues, 157–158  
 allowing users to add items to `StackedArray(T)` class, 184  
 allowing users to associate values with keys, 92–94, 123–130  
 allowing users to remove items from arrays, 13–17  
 allowing users to remove items from `CircularBuffer(T)` class, 203–204  
 allowing users to remove items from collections, 42–49  
 allowing users to remove items from doubly linked lists, 68–75  
 allowing users to remove items from `QueuedLinkedList(T)` class, 166–167  
 allowing users to remove items from queues, 158–159  
 allowing users to remove items from `StackedArray(T)` class, 180, 185  
 allowing users to remove keys, 95–98, 130–134

## V

values  
 arrays of, 3  
`ValueCollection` class, 437  
`ValueMember` property (`ComboBox` control), 540–541  
`Values` property, 99, 137, 305–306  
 variable Operation using named method (example), 219–220  
`var` keyword, 451  
 Visual Basic (Microsoft)  
 implementing `Item` property with, 20  
 lambda expressions in, 221  
`var` keyword in, 451  
 Visual C# (Microsoft), `var` keyword in, 451  
`void AddAfter(LinkedListNode(T) node, LinkedListNode(T) newNode)`, 273  
`void AddBefore(LinkedListNode(T) node, LinkedListNode(T) newNode)`, 274  
`void AddFirst(LinkedListNode(T) node)`, 275–276  
`void AddLast (LinkedListNode(T) node)`, 276  
`void AddRange(IEnumerable(T) collection)`, 224  
`void Add(T item)` (`List(T)` class), 223–224  
`void Add(TKey key, TValue value)`, 297  
`void Clear()`, 285, 290, 298, 331  
`void Enqueue(T item)`, 285  
`void ExceptWith(IEnumerable(T) other)`, 337–338  
`void ForEach(Action(T) action)`, 225–228

`void Insert(int index, T item)`, 231  
`void InsertRange(int index, IEnumerable collection)`, 231  
`void IntersectWith(IEnumerable(T) other)`, 333  
`void OnClear()` and `void OnClearComplete()`, 318  
`void OnGet()`, 325  
`void OnInsertComplete(Object key, Object value)`, 325  
`void OnInsert(int index, Object value)`, 319  
`void OnInsert(Object key, Object value)`, 325  
`void OnRemoveComplete(Object key, Object value)`, 326  
`void OnRemove(Object key, Object value)`, 326  
`void OnSetComplete(Object key, object oldValue, object newValue)`, 327  
`void OnSet(Object key, object oldValue, object newValue)`, 327  
`void OnValidate(object key, object value)`, 324–325  
`void OnValidate(object value)`, 317  
`void Push(T item)`, 289  
`void RemoveAt(int index)`, 230  
`void RemoveFirst()`, 278  
`void RemoveLast()`, 278–279  
`void RemoveRange(int index, int count)`, 230  
`void Reverse()`, 272  
`void Reverse(int index, int count)`, 272  
`void SetAll(bool value)`, 309  
`void Set(int index, bool value)`, 308  
`void Sort()`, 236  
`void Sort(Comparison(T) comparison)`, 236–239  
`void Sort(IComparer(T) comparer)`, 240–241  
`void Sort(int index, int count, IComparer(T) comparer)`, 242–246  
`void TrimExcess()`, 272, 288, 338  
`void UnionWith(IEnumerable(T) other)`, 337

## W

`WaitAll` method, 477  
`WaitAny` method, 477  
`WaitHandle` class, 476, 477  
`Wait` method, 479  
 websites, for further information  
`Dispose` method, 352  
 downloading code samples, xvii  
`HierarchicalDataTemplate`, 603  
`LINQ` data sources, 442  
`where` clause (LINQ), 458–459  
 Windows Forms  
 binding with `BindingSource` object, 581–582  
`BindingList(T) class`, using, 574  
`BindingSource` class, 575–576  
 binding with `ComboBox` control, 576–577  
 binding with `DataGridView` control/`IBindingListView`, 580  
 binding with `DataGridView`/`IBindingList` control, 578–580  
 binding with `IBindingList`, 578–580  
 binding with `ListBox` control, 577–578

Windows Forms (*continued*)  
IBindingList Interface, implementing. *See* IBindingList  
Interface  
IBindingListView interface, implementing. *See*  
IBindingListView interface (Windows Forms)  
simple binding, 539–540  
two-way data binding. *See* two-way data binding  
WinFormsBindingList(T) class, 547, 550, 559–560  
WinFormsBindingListView(T) class, 562  
WPF (Windows Presentation Foundation)  
ComboBox class and, 598  
ListBox class and, 600  
ListView class and, 601  
Treeview class and, 603  
using collections in, 583

## X

XmlSerializer, 513  
Xor method (BitArray class), 314–315

## About the Author

**Calvin Janes** is currently a Principal Software Consultant with over 20 years of professional experience. He has been hooked on programming ever since he wrote a 1,000-line BASIC program in third grade. Since then, he has worked on everything from 3D graphics to developing software for helicopters and control systems. When not working, he enjoys tinkering with microcontrollers and robotics, and exploring how they can be applied to the medical field. He plays several instruments and received an Emergency Medical Technician (EMT-B) certification while working as a volunteer firefighter.

