Data Binding

Data binding is a connection between the User Interface and a business object or other data provider . The User Interface object is called the ***target***, the provider of the data is called the ***source***.

Data-binding assists with the separation of the User Interface level of your application from the other layers of your application (business objects, data, and so forth). This separation of responsibility is further reinforced by decoupling the UI target from its source through the use of a Binding object.

The binding object can be thought of as a black box with a universal connectors on one side for the target and on the other side for the source. There are switches on top, the most important of which is the Data Binding Mode switch which determines which way the data will flow.

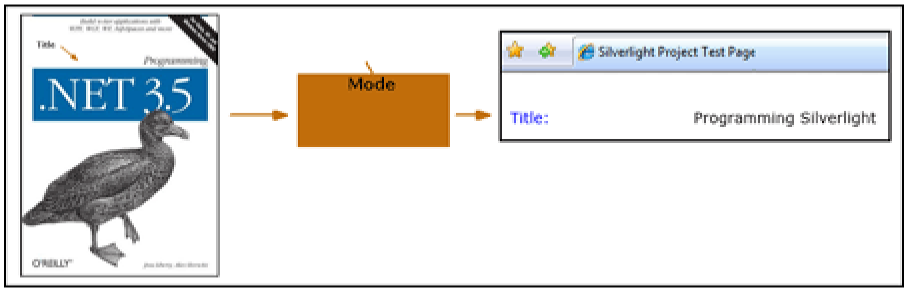
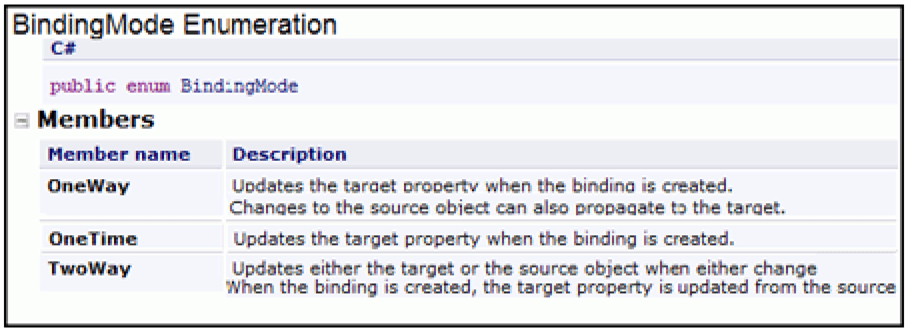


Figure 2-1. The Binding Object as a universal connector between target and source

Data Binding Modes

The Mode (which is of type BindingMode) is an enumeration with three possible values as shown here,



BindingMode Enumeration

C#

```
public enum BindingMode
```

Members

Member name	Description
OneWay	Updates the target property when the binding is created. Changes to the source object can also propagate to the target.
OneTime	Updates the target property when the binding is created.
TwoWay	Updates either the target or the source object when either change When the binding is created, the target property is updated from the source

Figure 2-2. The BindingMode Enumeration Documentation (Excerpt)

OneTime binding sets the target and then the binding is completed. This is great for displaying data that rarely or never changes

OneWay binding sets the target and keeps it up to date as the source changes. This is great for displaying data that the user is not permitted to change.

TwoWay binding sets the target and keeps the target up to date as the source changes and keeps the source up to date as the user changes the target or changes something else in the application that will cause your application to change the source.

If you were building an on line bookstore and were displaying information about a book, you might use OneTime binding for the Title and Author (once you get them, they are not going to change) and OneWay binding for the price (someone might mark down the price during the day) but you'd want TwoWay binding for the Quantity on hand.

The target of your binding can be any public property of virtually any CLR object (for the minor restrictions on this, see the Afterward).

You can see this by building a small example; but even in this stripped down exercise, we'll keep to a basic 3-tier approach being sure to keep a relatively strict separation among

- User Interface Layer
- Business Layer
- Persistence Layer.

The User Interface layer will consist of controls that we will obtain from the toolbox and use as offered for now, though a future tutorial will demonstrate how to change their appearance using styles and templates.

The Business Layer will be represented by a Book class

We will ignore the persistence layer for now, though this will be covered in great detail in future tutorials.

The Business object and INotifyPropertyChanged

Create a new Silverlight Application, which will create a Page.xaml and a Page.xaml.cs.

Add to the application a Book.cs file which will represent the business layer.

What separates a Silverlight Business object from one created for (e.g., ASP.NET) is that we want to have our business object participate in oneway or twoway binding with the UI layer.

For our control to be notified when the business object changes, the business object must implement the ***INotifyPropertyChanged*** interface.

This interface requires only one thing: that the class have an event of type PropertyChangedEventHandler (named PropertyChanged by convention), Implicit in supporting binding, however, is that your business object must, by convention, fire the PropertyChanged event when any property that is tied to a UI control is changed (and the most common way to change it, is to set its value).

Let's begin with a simplified version of the Book class,

```
public class Book : INotifyPropertyChanged
 {
     private string bookTitle;
     public event PropertyChangedEventHandler PropertyChanged;
     public string Title
     {
         get { return bookTitle; }
         set
         {
             bookTitle = value;
             if ( PropertyChanged != null )
             {
                PropertyChanged(this, new PropertyChangedEventArgs(“Title”);
             }
         }
     }
}
```

The first thing to notice is the using statement for System.ComponentModel which is necessary for the interface INotifyPropertyChanged.   The class states that it will fulfill this interface in the class declaration,

```
public class Book : INotifyPropertyChanged
```

What is required is that it have an instance of an event of type PropertyChangedEventHandler, which the pattern is to name PropertyChanged.

In this simplified version of the Book class, we have only one field, bookTitle, exposed as the property Title. We can't use the new C# 3.0 simplified property syntax:

```
public class Book : INotifyPropertyChanged
 {
     public event PropertyChangedEventHandler PropertyChanged;
     public string Title { get; set; }
  }
```

That is only legal when you are taking no action in get except returning the value and when you are taking no action in set except setting the value. Here, however, our setter is taking a section action; it is checking to see if anyone has registered with our event, and if so it is calling the registered method through the event's delegate.

Factoring Out Common Code As We Add Properties

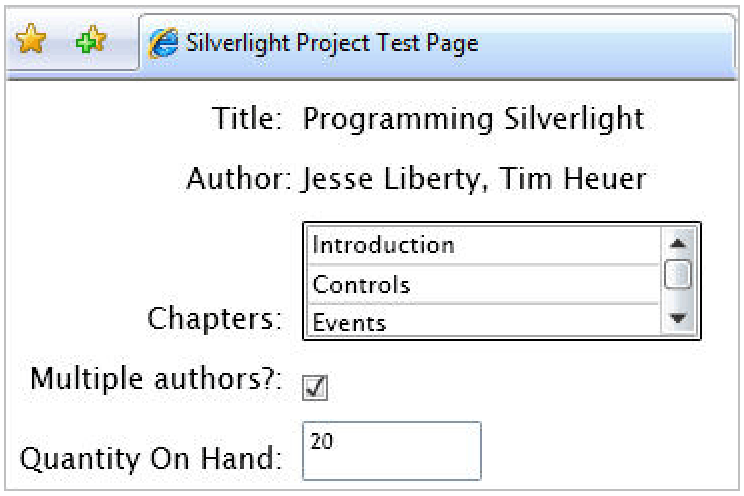
As we add more properties (bookAuthor, quantityOnHand, etc.) each setter will have to check to see if anyone has registered with the event and if so invoke the event. Cutting and pasting the code makes my skin crawl: such code does not scale well, is hard to maintain and is error prone. We'll factor out that responsibility into a method, and have each property call that method.

```
public class Book : INotifyPropertyChanged
    {
        private string bookTitle;
        private string bookAuthor;
        private int quantityOnHand;
        private bool multipleAuthor;
        private string authorURL;
        private string authorWebPage;
        private List<string> myChapters;


        // implement the required event for the interface
        public event PropertyChangedEventHandler PropertyChanged;

        public string Title
        {
            get { return bookTitle; }
            set
            {
                bookTitle = value;
                NotifyPropertyChanged("Title");
            }       // end set
        }           // end property

        public string Author
        {
            get { return bookAuthor; }
            set
            {
                bookAuthor = value;
                NotifyPropertyChanged("Author");
            }       // end set

        }

        public List<string> Chapters
        {
            get { return myChapters; }
            set
            {
                myChapters = value;
                NotifyPropertyChanged("Chapters");
            }
        }


        public bool MultipleAuthor
        {
            get { return multipleAuthor; }
            set
            {
                multipleAuthor = value;
                NotifyPropertyChanged("MultipleAuthor");
            }       // end set
        }


        public int QuantityOnHand
        {
            get { return quantityOnHand; }
            set
            {
                quantityOnHand = value;
                NotifyPropertyChanged("QuantityOnHand");
            }       // end set
        }


        // factoring out the call to the event
        public void NotifyPropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this,
                    new PropertyChangedEventArgs(propertyName));
            }

        }
    }
```

By factoring out the call to PropertyChanged, we can now pass in the name of any property and reuse this boilerplate to invoke the delegate.

Displaying the Data

To make a clean and easy display of the data, we'll create a form with two columns. On the left will be a prompt, on the right will be the data,



Title: Programming Silverlight

Author: Jesse Liberty, Tim Heuer

Chapters: Introduction, Controls, Events

Multiple authors?: ✓

Quantity On Hand: 20

Figure 2-3. Displaying a Book's Properties

For now we won't worry about how we get the book's information, we'll just focus on displaying it. All of the labels on the left we'll create as TextBlocks. The top two labels on the right will be text blocks as well, followed by a list box, a check box and then a TextBox. Note the difference between a TextBlock (essentially a label) and a TextBox which is used for data entry.

In the previous tutorial on controls you saw how to create the grid with rows and columns and how to place these controls, you also saw how to place text into the controls. Here is the code for all the controls leaving question marks where we need to do the databinding.

```
<UserControl x:Class="BookProperties.Page"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">

    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.RowDefinitions>
            <RowDefinition MaxHeight="30" />
            <RowDefinition MaxHeight="30" />
            <RowDefinition MaxHeight="70" />
            <RowDefinition MaxHeight="30" />
            <RowDefinition MaxHeight="40" />
            <RowDefinition MaxHeight="50" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition MaxWidth="150"/>
            <ColumnDefinition MaxWidth="200" />
        </Grid.ColumnDefinitions>

        <TextBlock x:Name="TitlePrompt" Text="Title:   "
            VerticalAlignment="Bottom"
            HorizontalAlignment="Right"
            Grid.Row="0"  Grid.Column="0" />
       <TextBlock x:Name="Title"
            Text="?"
            VerticalAlignment="Bottom"
```

```
              HorizontalAlignment="Left"
              Grid.Row="0" Grid.Column="1" />

          <TextBlock x:Name="AuthorPrompt" Text="Author: "
              VerticalAlignment="Bottom"
              HorizontalAlignment="Right"
              Grid.Row="1"  Grid.Column="0" />
         <TextBlock x:Name="Author"
              Text="?"
              VerticalAlignment="Bottom"
              HorizontalAlignment="Left"
              Grid.Row="1" Grid.Column="1" />

        <TextBlock x:Name="ChapterPrompt" Text="Chapters:  "
              VerticalAlignment="Bottom"
              HorizontalAlignment="Right"
              Grid.Row="2" Grid.Column="0"  />

          <ListBox x:Name="Chapters"
              ItemsSource="?"
              VerticalAlignment="Bottom"
              HorizontalAlignment="Left"
              Height="60" Width="200"
              Grid.Row="2" Grid.Column="1" />

          <TextBlock x:Name="MultipleAuthorPrompt"
              Text="Multiple authors?:  "
               VerticalAlignment="Bottom"
              HorizontalAlignment="Right"
              Grid.Row="3" Grid.Column="0"  />

          <CheckBox x:Name="MultipleAuthor"
              IsChecked="?"
              VerticalAlignment="Bottom"
              HorizontalAlignment="Left"
              Grid.Row="3" Grid.Column="1"/>

          <TextBlock x:Name="QOHPrompt"
          Text="Quantity On Hand:  "
          VerticalAlignment="Bottom"
          HorizontalAlignment="Right"
          Grid.Row="4" Grid.Column="0" />

          <TextBox x:Name="QuantityOnHand"
              Text="?"
            VerticalAlignment="Bottom"
              HorizontalAlignment="Left"
              Height="30" Width="90"
              Grid.Row="4" Grid.Column="1" />




      </Grid>
  </UserControl>
```

Binding To The TextBlocks

Let's start easy and bind to the first two text blocks. The pattern is to tell the text block that you are binding to a property, and to name the property. You also tell the Binding the mode of your binding (one way, two way or one time). The default is "one way" but it is good programming practice to make it explicit. This binding is done in the text field, and is surrounded by curly braces,

```
Text="{Binding Title, Mode=OneWay }"
```

Thus the first two rows look like this

```
<TextBlock x:Name="TitlePrompt" Text="Title:  "
     VerticalAlignment="Bottom"
     HorizontalAlignment="Right"
     Grid.Row="0"  Grid.Column="0" />

<TextBlock x:Name="Title"
     Text="{Binding Title, Mode=OneWay }"
     VerticalAlignment="Bottom"
     HorizontalAlignment="Left"
     Grid.Row="0" Grid.Column="1" />

  <TextBlock x:Name="AuthorPrompt" Text="Author: "
     VerticalAlignment="Bottom"
     HorizontalAlignment="Right"
     Grid.Row="1"  Grid.Column="0" />

<TextBlock x:Name="Author"
     Text="{Binding Author, Mode=OneWay }"
     VerticalAlignment="Bottom"
     HorizontalAlignment="Left"
     Grid.Row="1" Grid.Column="1" />
```

We don't' yet know which object's Title and Author properties we are binding to, but we expect to have an object with these properties.

The same is done for the other controls. ListBox has its ItemSource bound, but in this case it expects to be bound to a collection (specifically to an IEnumerable).

```
<ListBox x:Name="Chapters"
    ItemsSource="{Binding  Chapters, Mode=OneWay}"
    VerticalAlignment="Bottom"
    HorizontalAlignment="Left"
    Height="60" Width="200"
    Grid.Row="2" Grid.Column="1" />
```

The Checkbox expects to bind to an object that will resolve to a Boolean.

```
<CheckBox x:Name="MultipleAuthor"  IsChecked="{Binding MultipleAuthor,
Mode=TwoWay}"
    VerticalAlignment="Bottom"
    HorizontalAlignment="Left"
    Grid.Row="3" Grid.Column="1"/>
```

DataContext

When you DataBind you tell the target some of what it needs to know at design time (e.g., you might tell a TextBlock that it will be displaying the Title of a book) but you don't want to tell it exactly ***which*** book's title it will be displaying. The DataContext is the specific book, chosen at run time, and assigned to the DataContext property of the Framework Element (in this case the TextBlock) so that it knows "Oh, I get the Title from ***this*** book" More on this shortly.

The DataContext can be raw data, but far more common is to assign an object of type Binding. A Binding object knows how to get the data needed by the target from the source. It is our universal connector with the Mode switch on top.

Another advantage of using a DataContext is that it allows elements to inherit information about the data source from the parent element. As an example, you can set the data source for a grid, and all the controls in that grid are free to use that DataContext without having their own DataContext explicitly set.

You can replace

```
Title.DataContext = currentBook;
Author.DataContext = currentBook;
Chapters.DataContext = currentBook;
MultipleAuthor.DataContext = currentBook;
QuantityOnHand.DataContext = currentBook;
```

With this one line

```
LayoutRoot.DataContext = currentBook;
```

The Logic of Data Binding Step By Step

1. Create a target object (e.g, TextBlock) and identify the Property that will be Bound
2. Identify the source and a property on that source whose value you'll bind to the target
3. Map the Target to the Source's property using a DataContext

It's easier than it sounds, especially the second time.

We want our Silverlight application to be able to display the details of ***any*** book that might be chosen, so we don't want to hardcode the values. Data Binding allows us to set up the properties of the book, and then pick the book we want at run time.

To see this, we'll add a new button, ***Change*** that will toggle between displaying two different book objects

```
<Button x:Name="Change"
Content="Change Book"
Height="30" Width="80"
HorizontalAlignment="Right"
Grid.Row="5" Grid.Column="0"  />
```

As you can see, there's nothing special about the button; the magic is in the code. We accomplish this small miracle in small steps. First, pull the reference to the Book out into a member variable, second create three new member variables

```
private Book b1;
private Book b2;
private Book currentBook;
```

You'll initialize the two books in memory and have the first referred to by the identifier **b1** and the second referred to by **b2**, and currentBook will toggle between them. A hack, I admit, but it will work and will keep the code simple.

The premise here is to mimic the idea that in a "real" system you'll be picking one book not from two, but from many,

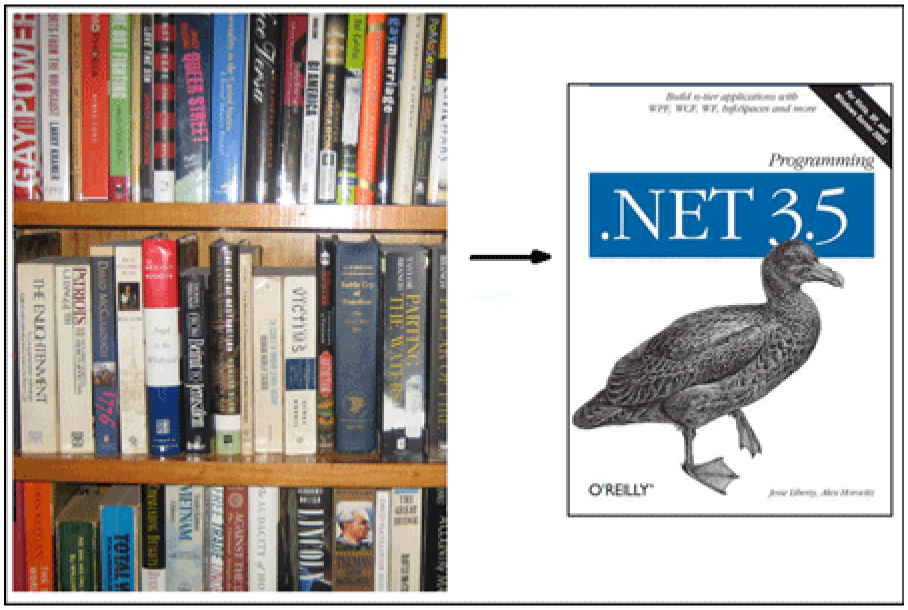


Figure 2-4. Extracting one book from a library of possibilities

You'll initialize each book once, when we start the program, and then swap between them.

```
b1 = new Book();
InitializeBleak(b1);
currentBook = b2 = new Book()
InitializeProgramming(b2);
```

The initialization sets the book's properties.

```
private void InitializeProgramming(Book b)
 {
     b.Title = "Programming Silverlight";
     b.Author = "Jesse Liberty, Tim Heuer";
     b.MultipleAuthor = true;
     b.QuantityOnHand = 20;
     b.Chapters = new List<string>()
         { "Introduction", "Controls", "Events", "Styles" };
 }

 private void InitializeBleak(Book b)
 {
     b.Title = "Bleak House";
     b.Author = "Charles Dickens";
     b.MultipleAuthor = false;
     b.QuantityOnHand = 150;
     b.Chapters = new List<string>()
     {
         "In Chancery",
         "In Fashion",
         "A Progress",
         "Telescoopic Philanthropy",
         "A Morning Adventure",
         "Quite at Home",
         "The Ghosts Walk",
         "Covering Sins",
         "Signs and Tokens",
         "The Law Writer"
     };
 }
```

TextBox

In addition to adding a CheckBox we add a second Read/Write field, the TextBox, which will display and allow the user to update the number of copies of the book on hand.

```
<TextBox x:Name="QuantityOnHand"
    Text="{Binding QuantityOnHand, Mode=TwoWay}"
    VerticalAlignment="Bottom"
    HorizontalAlignment="Left"
    Height="30" Width="90"
    Grid.Row="4" Grid.Column="1" />
```

In our design, each book knows how many copies of itself are in the store; there certainly are other ways of designing this.

ListBox and Binding to a List

Finally, we come to the Chapters. The Book object defines the Chapters as a list of strings

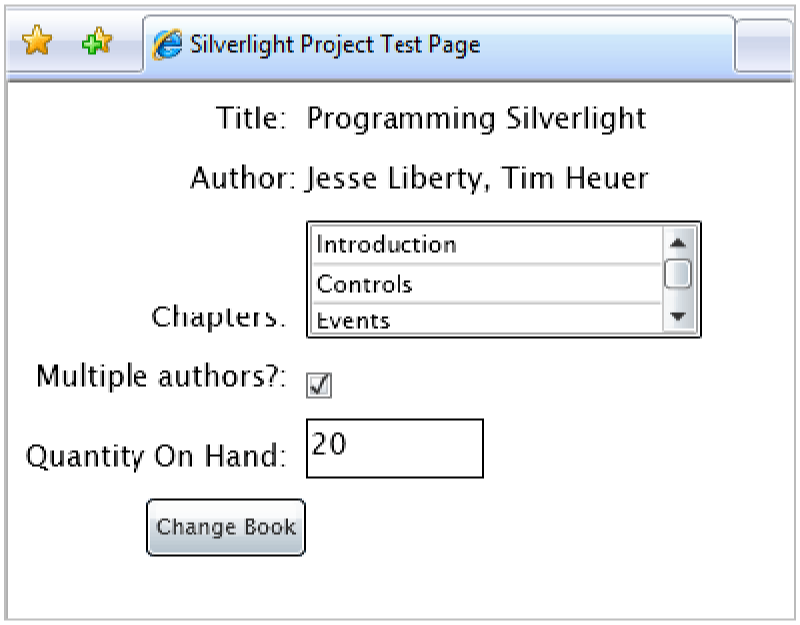
```
public List<string> Chapters
{
    get { return myChapters; }
    set
    {
        myChapters = value;
        NotifyPropertyChanged("Chapters");
    }
}
```

When the book is displayed, the chapters will be displayed in the list box. When a new book is chosen, that new book's chapters will be displayed. You ***could*** iterate through the new book's chapters and create a new set of ListItems, but again that is error prone and doesn't scale well. Once again, Data-binding is a more satisfactory solution.

Since the DataSource for the Title and Author (the current book) will also have the correct list of Chapters, you can assign the Chapter property of the Book to the ItemSource property of the List Box, and as the DataSource is changed, the chapters will be changed appropriately,

```
<ListBox x:Name="Chapters"
    ItemsSource="{Binding  Chapters, Mode=OneWay}"
    VerticalAlignment="Bottom"
    HorizontalAlignment="Left"
    Height="60" Width="200"
    Grid.Row="2" Grid.Column="1" />
```

The next two images show the new details of the two books displayed as the user switches among them.



Title: Programming Silverlight

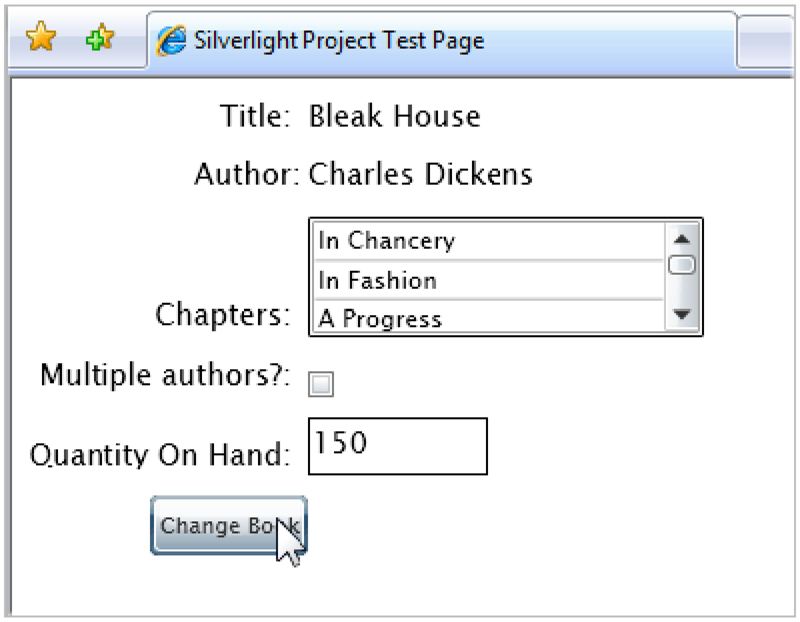
Author: Jesse Liberty, Tim Heuer

Chapters: Introduction, Controls, Events

Multiple authors?: ✓

Quantity On Hand: 20

Figure 2-5. The First Book Displayed



Title: Bleak House

Author: Charles Dickens

Chapters: In Chancery, In Fashion, A Progress

Multiple authors?: ☐

Quantity On Hand: 150

Figure 2-6. The Second Book Displayed