O'REILLY®

~~Fourth~~ Fifth Edition

This is part of an early release preview of the 5th edition of Head First C# by Andrew Stellman and Jenny Greene. We'll release the final version of this PDF when the book is published in late 2023.

Head First C#

A Learner's Guide to Real-World Programming with C# and .NET Core

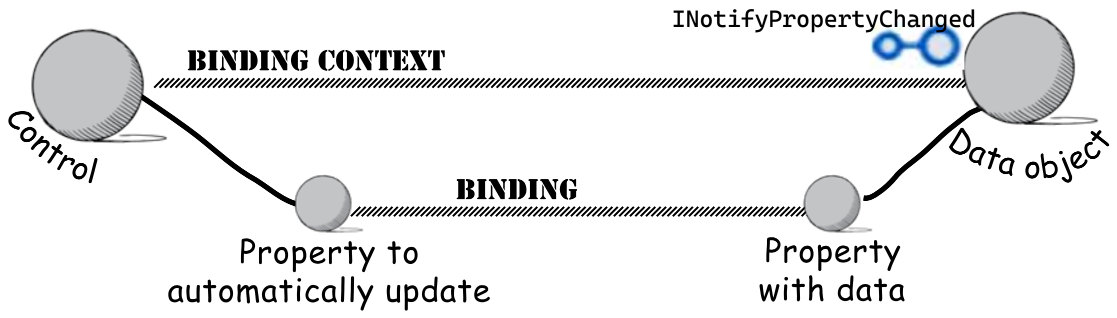
Andrew Stellman
& Jennifer Greene

This is the .NET MAUI project from Chapter 7. You'll create a simple app to help you learn about data binding in MAUI, then modify the Beehive Management System to use data binding.

A Brain-Friendly Guide

Data binding updates MAUI controls automatically

Here's a great example of a real-world use case for an interface: **data binding**. Data binding is a really useful feature in MAUI that lets you set up your controls so their properties are automatically set based on a property in an object, and when that property changes your controls' properties are automatically kept up to date.



1. **Data binding starts with a data object.**
If you want your controls to automatically update themselves, they need to have some data to use. A **data object** is an object with data that you want to automatically send to your controls.

2. **The binding context tells your controls where to find the data.**
The goal of data binding is to feed data to the controls on your page automatically, so it makes sense that you they would need some place to get that data. The binding context is the object that contains the data you'll use in your controls.

   You can set the binding context for a control by setting its **BindingContext property** to point to an object that it will get data from. If you set the binding context for a control that contains other controls—like a ScrollView or a Grid—then the nested controls get the same binding context

3. **The XAML includes bindings that tell the controls what data to bind to.**
Once oyu have the data object created and the binding context set, you need to tell your controls to look for that data and use it to update themselves. You do this by setting up a **binding**, special text inside the XAML property value that looks like this: **`{Binding PropertyToBindTo}`**

Brain Power

The data object has a **specific job**: it needs to notify the binding context any time its data has changed, so the controls can update themselves. How do you think you'll tell your data object to do a specific job?

Add data binding to the default MAUI app

Let's get use data binding to make the cute robot in the default MAUI app can tell you how it's feeling.

1 Create a net .NET MAUI app and add a new class with data to bind to.
Create a new .NET MAUI app, then **add a new class called Moods**. It has a property called CurrentMood that's updated any time its UpdateMood method is called:

Do this!

```
class Moods
{
    public string CurrentMood { get; private set; }

    public Moods()
    {
        UpdateMood();
    }

    public void UpdateMood()
    {
        switch (Random.Shared.Next(1, 4))
        {
            case 1:
                CurrentMood = "I'm happy!";
                break;
            case 2:
                CurrentMood = "Oh so sad.";
                break;
            default:
                CurrentMood = "Feeling good.";
                break;
        }

        SemanticScreenReader.Announce(CurrentMood);
    }
}
```

When UpdateMood is called, it picks a random number from 1 to 3 and uses it to choose a mood.

Let's make it accessible! The screen reader will announce every mood change.

2 Set the binding context for your page to a new instance of Moods.
One easy way to set the binding context for all of the controls on your page is to set the BindingContext property of the page object in the MainPage constructor.

```
public partial class MainPage : ContentPage
{
    int count = 0;

    Moods moods = new Moods();

    public MainPage()
    {
        InitializeComponent();

        BindingContext = moods;
    }
```

Create a new instance of the Moods class.

Add a statement to the page's constructor to set its BindingContext property so it contains a reference to the Moods object.

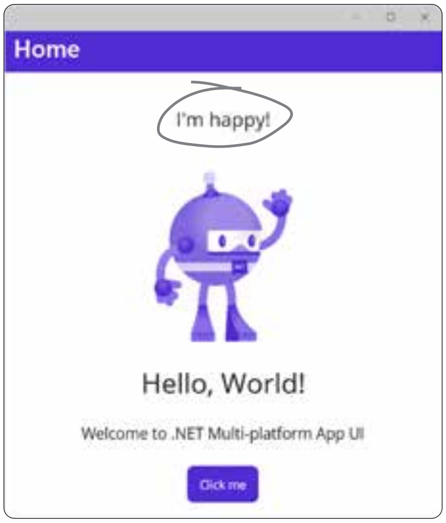
3 Add a new Label with a binding to the top of the page.
Add the this XAML for a Label control just below the opening <VerticalStackLayout> tag:

```
<Label Text="{Binding CurrentMood}" FontSize="Medium"
       HorizontalOptions="Center" Margin="20"/>
```

4 Run your app!

Congratulations—your app now uses data binding! You created an instance of a data object, set the data context, and added a binding. Now the Label control uses that binding for its Text propery. The binding tells it to read its data from the CurrentMood property in the data context.

Try running your app several times to make sure that it's displaying a random mood each time.



5 Update the mood when the button is clicked.

The real power of data binding is being able to keep properties in the controls on your page up to date as the data object changes. **Modify the Click event handler method** to call the UpdateMood method on your Moods data object:

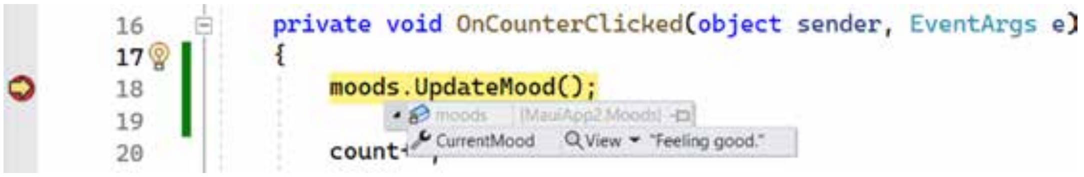
```
private void OnCounterClicked(object sender, EventArgs e)
{
    moods.UpdateMood();

    count++;

    if (count == 1)
        CounterBtn.Text = $"Clicked {count} time";
    else
        CounterBtn.Text = $"Clicked {count} times";

    SemanticScreenReader.Announce(CounterBtn.Text);
}
```

Calling UpdateMood will cause the CurrentMood to change to a new random mood every time the button is clicked.

6 Run the app again.

Hold on—something's not working. **The mood on the page isn't changing!** If you restart the app, it picks a new random mood. But clicking the button doesn't cause the page to get updated.

Try putting a breakpoint on the line that calls moods.UpdateMood and clicking the button. When the breakpoint breaks, hover over moods to see the value of the CurrentMood property:



```
private void OnCounterClicked(object sender, EventArgs e)
{
    moods.UpdateMood();

    count++;
```

```
moods    {MauiApp2.Moods}
    CurrentMood    View    "Feeling good."
```

Step over the method. The property value changes, but the page doesn't.

Why do you think the page doesn't update when the property changes?

Make Moods implement the INotifyPropertyChanged interface

A data object has a specific job: notifying the binding context any time its data has changed. Data objects can be any kind of object—they don't have to extend any specific class. We learned throughout this chapter that when you need an object to do a specific job, you make it implement an interface. In .NET MAUI data binding, data object is a job. So it makes sense that there would be an interface for it!

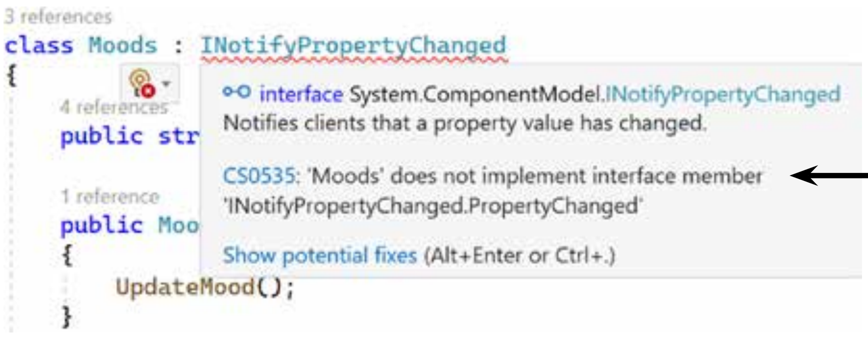
A data object that needs to update properties should implement the **INotifyPropertyChanged interface**. This interface does exactly what it sounds like—it's notifies the binding context any time a property changed.

The INotifyPropertyChanged interface is in the System.ComponentModel namespace. When you added the Moods.cs file to your project, Visual Studio may have added using statements at the top. Add this using statement to the file if it isn't already there:

Do this!

```
using System.ComponentModel;
```

Then modify the Moods class to **implement the INotifyPropertyChanged interface**. The interface is not implemented, so Visual Studio will warn you about a compiler error:



```
3 references
class Moods : INotifyPropertyChanged
{
    4 references
    public str
    1 reference
    public Moo
    {
        UpdateMood();
    }
```

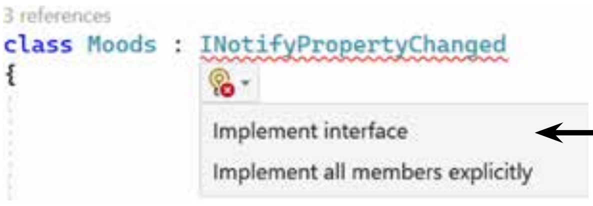
interface System.ComponentModel.INotifyPropertyChanged
Notifies clients that a property value has changed.

CS0535: 'Moods' does not implement interface member 'INotifyPropertyChanged.PropertyChanged'

Show potential fixes (Alt+Enter or Ctrl+.)

This compiler error happens when a class that implements an interface has not implemented all of its members.

Use the Quick Actions menu to implement the interface. Press Alt+Enter or ⌥↵ to show potential fixes **and choose "Implement interface"** from the context menu.



```
3 references
class Moods : INotifyPropertyChanged
{
```

When you the Quick Actions menu (Alt+Enter or ⌥↵) to implement an interface, Visual Studio automatically generates the interface members for you.

As soon as you choose that action from the menu, Visual Studio should add this line of code to your Moods class:

```
    public event PropertyChangedEventHandler PropertyChanged;
```

The INotifyPropertyChanged interface has a single member, an **event** called PropertyChanged.

We haven't talked about events yet. We'll give you all of the code that you need to use this event. You can learn more about events in our downloadable chapter on events and delegates: https://github.com/head-first-csharp/fifth-edition

Use the PropertyChanged event to make data binding work

This interface has a new keyword that we haven't shown yet: **`event`**. An event is a way for an object to ***let other objects know that something happened***. Buttons have a Clicked event, and you've written event handler methods for them: if you've hooked up an event handler method to a button, then when the user clicks the button, it **invokes** its Clicked event, which calls your event handler.

Now your Moods class has a PropertyChanged event, just like the Button class has a Clicked event.

Luckily, events are easy to use. **Add this method to your Moods class**:

Do this!

```
protected void OnPropertyChanged(string name)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

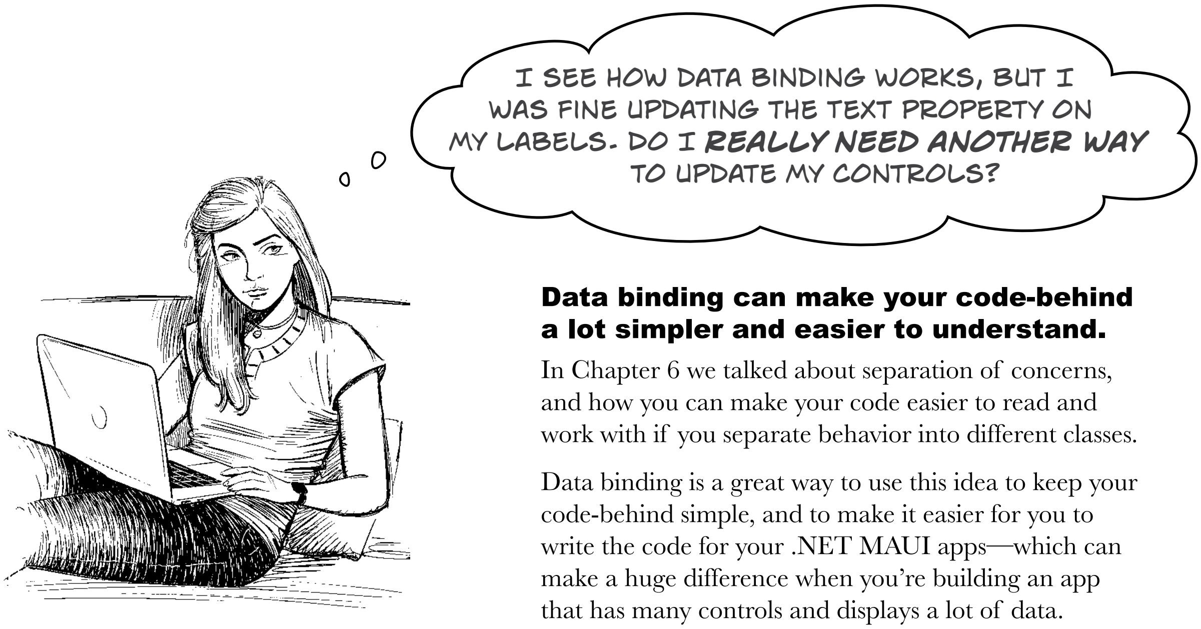
Now all you need to do to tell the binding context that the CurrentMood property has changed is to call this method and pass the string `"CurrentMood"` as an argument to invoke the PropertyChanged event.

Modify the Moods class to **add this statement to the very end of the UpdateMood method**:

```
OnPropertyChanged("CurrentMood");
```

Now run your app again. Click the button several times. It now works—the label text updates.

You added an event to your class, and added a method that uses the ?. operator to invoke the event to tell the binding context that a property was updated. That's all you need to know about events to finish the rest of the projects in this chapter!


I SEE HOW DATA BINDING WORKS, BUT I WAS FINE UPDATING THE TEXT PROPERTY ON MY LABELS. DO I REALLY NEED ANOTHER WAY TO UPDATE MY CONTROLS?

Data binding can make your code-behind a lot simpler and easier to understand.

In Chapter 6 we talked about separation of concerns, and how you can make your code easier to read and work with if you separate behavior into different classes.

Data binding is a great way to use this idea to keep your code-behind simple, and to make it easier for you to write the code for your .NET MAUI apps—which can make a huge difference when you're building an app that has many controls and displays a lot of data.

Sharpen your pencil

In the next exrecise you'll convert the Beehive Management System to use data binding. Here's new code-behind for it. Compare it with the code in Chapter 6 and write down what we changed.

```
public partial class MainPage : ContentPage
{
    private Queen queen = new Queen();

    public MainPage()
    {
        InitializeComponent();

        JobPicker.ItemsSource = new string[]
        {
            "Nectar Collector",
            "Honey Manufacturer",
            "Egg Care"
        };
        JobPicker.SelectedIndex = 0;

        Dispatcher.StartTimer(TimeSpan.FromSeconds(1.5), TimerTick);

        BindingContext = queen;
    }

    private bool TimerTick()
    {
        if (!this.IsLoaded) return false;

        WorkShiftButton_Clicked(this, new EventArgs());
        return true;
    }

    private void WorkShiftButton_Clicked(object sender, EventArgs e)
    {
        if (!queen.WorkTheNextShift())
            SemanticScreenReader.Default.Announce(OutOfHoneyButton.Text);
    }

    private void AssignJobButton_Clicked(object sender, EventArgs e)
    {
        queen.AssignBee(JobPicker.SelectedItem.ToString());
    }

    private void OutOfHoneyButton_Clicked(object sender, EventArgs e)
    {
        HoneyVault.Reset();
        queen = new Queen();
        BindingContext = queen;
    }
}
```

What changed in the constructor?

....

....

....

....

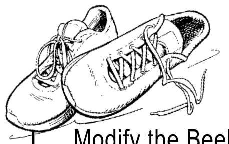
....

What changed in the rest of the code?

....

....

....



Exercise

Modify the Beehive Management System to use data binding, using the code in the example that we just gave you.

Modify the Queen class to implement INotifyPropertyChanged

- Add the using System.ComponentModel directive to Queen.cs if it's not already there.
- Modify the Queen class declaration to add `, INotifyPropertyChanged` and use the Quick Action menu to implement the interface so it adds the PropertyChanged event.
- Add the OnPropertyChanged method to your queen class (this is the same method that you added to Moods cs).
- Add two new properties to Queen:

```
public bool HiveIsRunning { get; private set; } = true;
public bool OutOfHoney { get { return !HiveIsRunning; } }
```

- Add code to the end of the UpdateStatusReport method to notify the binding context that properties have changed:

```
OnPropertyChanged("StatusReport");
OnPropertyChanged("CanAssignWorkers");
OnPropertyChanged("HiveIsRunning");
OnPropertyChanged("OutOfHoney");
```

- Replace the last line of the WorkTheNext method with this code to update the HiveIsRunning property:

```
HiveIsRunning = base.WorkTheNextShift();
return HiveIsRunning;
```

Modify the XAML to bind property values to properties in the Queen class

Modify the XAML to bind AssignJobButton's IsEnabled property, WorkShiftButton's IsVisible property, OutOfHoneyButton's IsVisible, and StatusReport's Text property to StatusReport.

```
<Button x:Name="AssignJobButton" Text="Assign this job to a bee"
        FontSize="Medium" Clicked="AssignJobButton_Clicked"
        SemanticProperties.Hint="Assigns the job to a worker bee"
        IsEnabled="{Binding CanAssignWorkers}" />

<Button x:Name="WorkShiftButton" Grid.Row="2" Text="Work the next shift"
        FontSize="Large" Clicked="WorkShiftButton_Clicked"
        SemanticProperties.Hint="Makes the bees work the next shift"
        IsVisible="{Binding HiveIsRunning}" />

<Button x:Name="OutOfHoneyButton" Grid.Row="2" Text="The hive is out of honey"
        BackgroundColor="Red" FontSize="Large" IsVisible="{Binding OutOfHoney}"
        Clicked="OutOfHoneyButton_Clicked"
        SemanticProperties.Hint="Restarts the Beehive Management System" />

<Label x:Name="StatusReport" Grid.Column="1" Grid.Row="1" Grid.RowSpan="2"
       VerticalOptions="FillAndExpand" FontSize="Small"
       BackgroundColor="Black" TextColor="Yellow" Padding="10"
       Text="{Binding StatusReport}" />
```

The "Out of honey" button's IsVisible property is already set in the XAML, so make sure you change it.

Sharpen your pencil Solution

```
public partial class MainPage : ContentPage
{
    private Queen queen = new Queen();

    public MainPage()
    {
        InitializeComponent();

        JobPicker.ItemsSource = new string[]
        {
            "Nectar Collector",
            "Honey Manufacturer",
            "Egg Care"
        };
        JobPicker.SelectedIndex = 0;

        Dispatcher.StartTimer(TimeSpan.FromSeconds(1.5), TimerTick);

        BindingContext = queen;
    }

    private bool TimerTick()
    {
        if (!this.IsLoaded) return false;

        WorkShiftButton_Clicked(this, new EventArgs());
        return true;
    }

    private void WorkShiftButton_Clicked(object sender, EventArgs e)
    {
        if (!queen.WorkTheNextShift())
            SemanticScreenReader.Default.Announce(OutOfHoneyButton.Text);
    }

    private void AssignJobButton_Clicked(object sender, EventArgs e)
    {
        queen.AssignBee(JobPicker.SelectedItem.ToString());
    }

    private void OutOfHoneyButton_Clicked(object sender, EventArgs e)
    {
        HoneyVault.Reset();
        queen = new Queen();
        BindingContext = queen;
    }
}
```

What changed in the constructor?

The constructor no longer calls the UpdateStatusAndEnableAssignButton method. Instead, it sets the binding context so it points to a reference to the Queen object in the queen field.

The updated code-behind in MainPage.xaml.cs is a lot shorter—and easier to understand!—because it doesn't have to manually set properties on controls.

The original Beeihve Management System code reset the system by creating a new Queen object. We just need to update the binding context to get all of the page controls to reflect that change.

What changed in the rest of the code?

UpdateStatusAndEnableAssignButton has been deleted. The code no longer sets the StatusReport.Text, AssignJobButton.IsEnabled, WorkShiftButton.IsVisible, or OutOfHoneyButton.IsVisible buttons. The OutOfHoneyButton Clickede event handler method updates the binding context.

Exercise

We gave you all of the changes to the XAML. Here are the changes to the Queen class. It's been modified to implement the INotifyPropertyChanged interface, and the OnPropertyChanged button was added.

```
class Queen : Bee, INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string name)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }
```

This works exactly like the code that you added to the Moods class when you added the label to the default MAUI app.

Two new properties were added, which are used to bind to properties:

```
    public bool HiveIsRunning { get; private set; } = true;
    public bool OutOfHoney { get { return !HiveIsRunning; } }
```

The HiveIsRunning property defaults to true. If the hive runs out of honey, she flips it to false – which causes OutOfHoney to return true.

The UpdateStatusReport method was modified to call OnPropertyChanged every time the Queen updated the report:

```
    private void UpdateStatusReport(bool allWorkersDidTheirJobs)
    {
        StatusReport = $"Vault report:\n{HoneyVault.StatusReport}\n" +
        $"\nEgg count: {eggs:0.0}\nUnassigned workers: {unassignedWorkers:0.0}\n" +
        $"{WorkerStatus("Nectar Collector")}\n{WorkerStatus("Honey Manufacturer")}" +
        $"\n{WorkerStatus("Egg Care")}\nTOTAL WORKERS: {workers.Length}";

        if (!allWorkersDidTheirJobs)
            StatusReport += "\nWARNING: NOT ALL WORKERS DID THEIR JOBS";

        OnPropertyChanged("StatusReport");
        OnPropertyChanged("CanAssignWorkers");
        OnPropertyChanged("HiveIsRunning");
        OnPropertyChanged("OutOfHoney");
    }
```

These are the four properties that you added bindings to when you modified the XAML. Now the app will update them every time the status report is regenerated at the end of the shift.

The WorkTheNextShift method was modified to set the HiveIsRunning property before returning:

```
    public override bool WorkTheNextShift()
    {
        eggs += Constants.EGGS_PER_SHIFT;
        bool allWorkersDidTheirJobs = true;
        foreach (Bee worker in workers)
        {
            if (!worker.WorkTheNextShift())
                allWorkersDidTheirJobs = false;
        }
        HoneyVault.ConsumeHoney(unassignedWorkers * Constants.HONEY_PER_UNASSIGNED_WORKER);
        UpdateStatusReport(allWorkersDidTheirJobs);

        HiveIsRunning = base.WorkTheNextShift();
        return HiveIsRunning;
    }
```

All of the Bee subclasses call the base class's WorkTheNextShift method, which returns false if the hive ran out of honey. This change uses that value to set the HiveIsRunning property before returning it.