O'REILLY®

Fourth Edition

Head First

C#

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

Andrew Stellman
& Jennifer Greene

A Brain-Friendly Guide

Controls drive the mechanics of your user interfaces

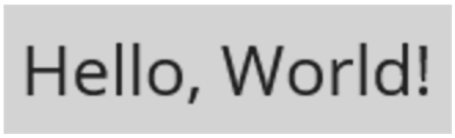
In the last chapter, you built a game using Button and Label **controls**. There are a lot of different ways that you can use controls, and the choices you make about what controls to use can really change your app. Does that sound weird? It's actually ***really similar to the way we make choices about mechanics in game design***. If you're designing a tabletop game that needs a random number generator, you can choose to use dice, a spinner, or cards. If you're designing a platformer, you can choose to have your player jump, double jump, wall jump, or fly (or do different things at different times). The same goes for apps: if you're designing an app where the user needs to enter a number, you can choose from different controls to let them do that… ***and that choice affects how your user experiences the app***.

Meet some of the controls you'll use in this book

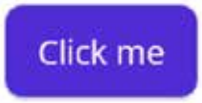
Most of the chapters in this book feature a .NET MAUI projcet. We included them so you can go beyond console apps and start learning how to build visual apps. In those projects, you'll use many different controls to build each app's the **user interface** (or **UI**)—or the way the window is laid out so the user can interact with it—of each app.

Here are the controls you've seen so far.

An Image control does exactly what you'd expect it to do—it displays an image. In this case, it's displaying the image in a file called dotnet_bot.png.



A Label displays text. You can set the font size, color, spacing, and text decorations (like italics or boldface).



A Button control shows a clickable button. It can call a method when you click it, and you can set or change its text.

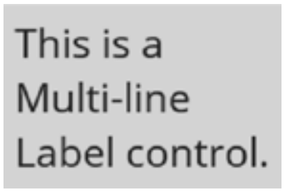
The user interface, or UI, is the part of the app that your user interacts with. In a console app, the UI is made up of text, and the user uses the keyboard to interact with it. In a .NET MAUI app, the UI is built using controls.

Other controls you'll use in this book

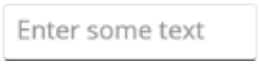
Controls are common user interface components, and they serve as the building blocks of your UI. The choices you make about what controls to use change the mechanics of your app.

Most of the chapters in this book contain a .NET MAUI project. You'll use various controls to build the UI for each of those apps. Here are a few of the ones that you'll use.

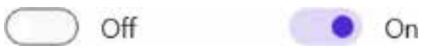
We can borrow the idea of mechanics from video games to understand our options, so we can make great choices for any of our own apps—not just games.



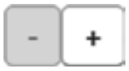
An Label can include multiple lines, which are separated by line breaks so it knows how to split them up.



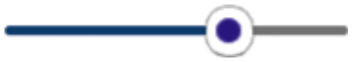
An Entry control lets your user enter text. It displays a placeholder, or lighter-colored text that gives the user some information about what they should type.

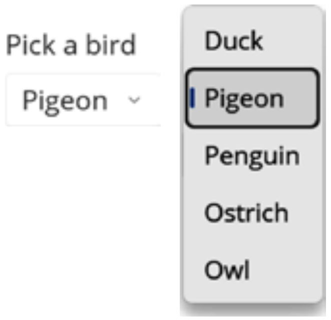


A Switch is a horizontal button that lets the user toggle (or switch back and forth) between two states, in this case on and off.

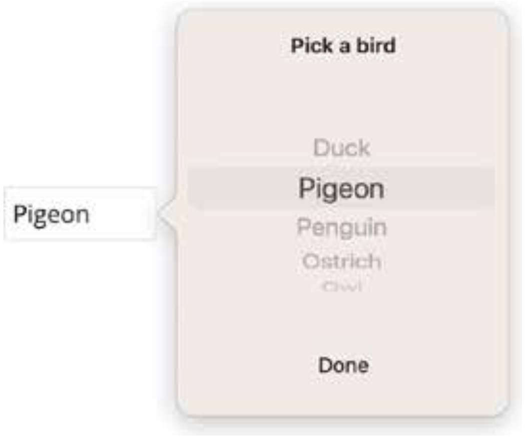


These are two different controls that let users enter numbers. A Stepper (on the left) presents the user with two buttons to increment or decrement—add or remove one—to a value. A Slider (on the right) lets the user slide back and forth to choose a decimal number.



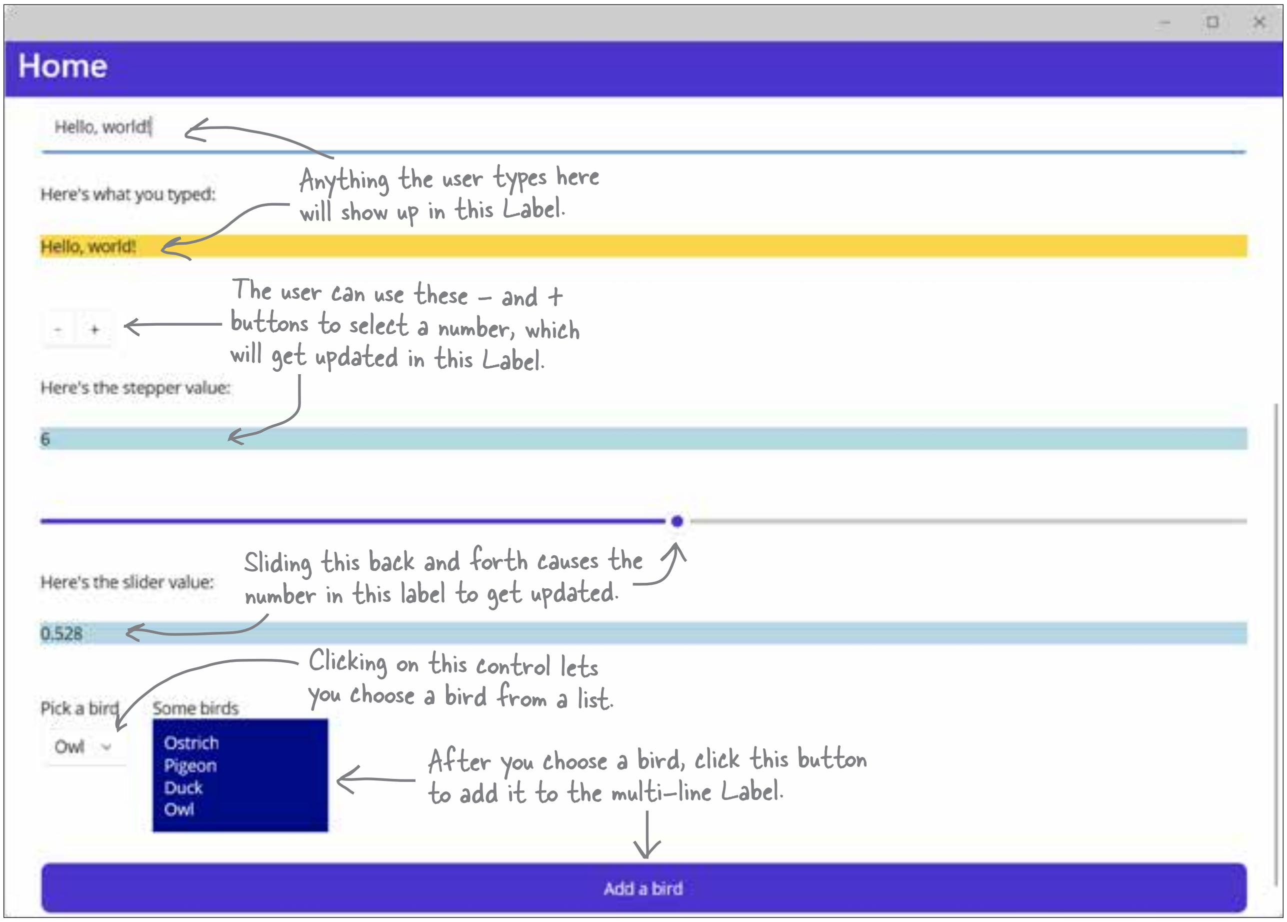


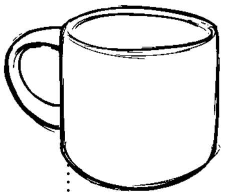
A Picker lets the user choose an item from a list. It looks a little different in Windows (on the left) and macOS (on the right), but both versions function in exactly the same way.



Build a .NET MAUI app to experiment with controls

You've probably seen most the controls we just showed you (even if you didn't know all of their official names). Now let's **create a .NET MAUI app** to get some practice using some of them. The app will be really simple—the user will use controls to enter values, and the app will display those values.


Home
Hello, world!
Here's what you typed:
Hello, world!
Here's the stepper value:
6
Here's the slider value:
0.528
Pick a bird
Owl
Some birds
Ostrich
Pigeon
Duck
Owl
Add a bird
Anything the user types here will show up in this Label.
The user can use these – and + buttons to select a number, which will get updated in this Label.
Sliding this back and forth causes the number in this label to get updated.
Clicking on this control lets you choose a bird from a list.
After you choose a bird, click this button to add it to the multi-line Label.



Relax

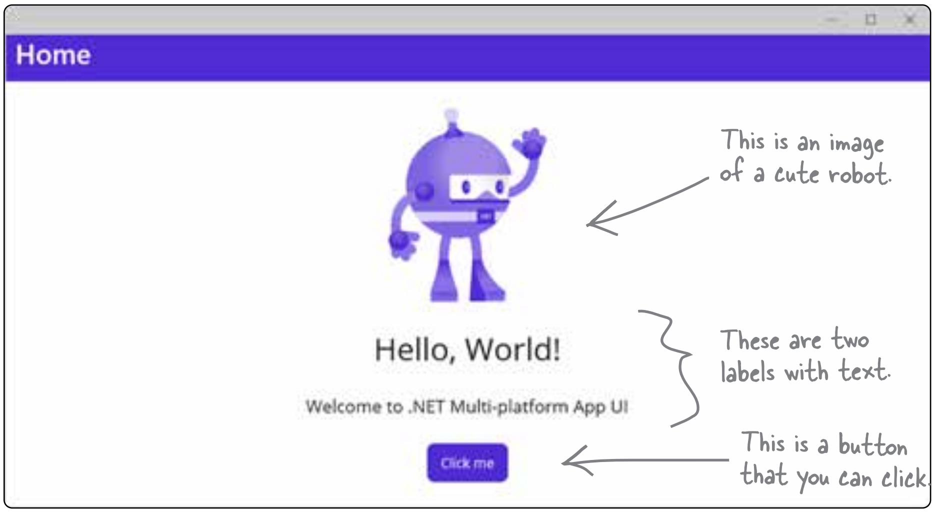
Don't worry about memorizing the XAML in this project. You'll pick it up throughout the book.

This Do this! and these exercises are all about getting some practice using XAML to build a UI with controls. You can always refer back to it when we use these controls in projects later in the book.

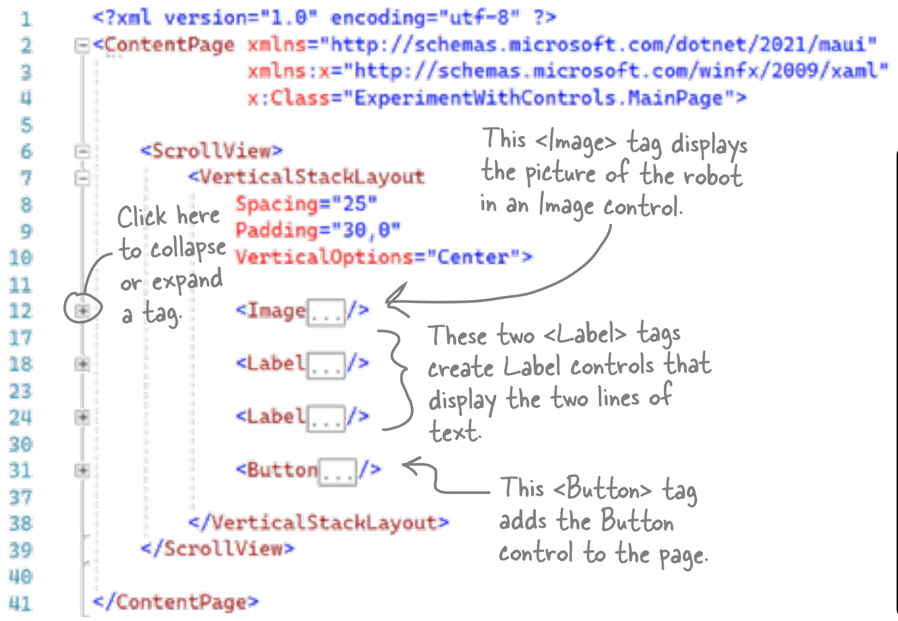
Create a new app to experiment with controls

Do this!

Go back to Visual Studio and **create a new .NET MAUI project**, just like you did in Chapter 1. Name your project ***ExperimentWithControls***. Run your new .NET MAUI app. It will pop up a window with a picture of a cute robot, text that says *Hello, World!*, and smaller text that says *Welcome to .NET Multiplatform APP UI*, and finally a button with the label *Click me*.


Home
Hello, World!
Welcome to .NET Multi-platform App UI
Click me
This is an image of a cute robot.
These are two labels with text.
This is a button that you can click.

Now go back to Visual Studio and **double-click the file MainPage.xaml** to open it. Use the buttons in the left margin that look like a minus sign in a square to collapse the <Image>, <Label>, and <Button> tags. Each of those tags corresponds to one of the controls in your app.



```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ExperimentWithControls.MainPage">

    <ScrollView>
        <VerticalStackLayout
            Spacing="25"
            Padding="30,0"
            VerticalOptions="Center">

            <Image.../>

            <Label.../>

            <Label.../>

            <Button.../>

        </VerticalStackLayout>
    </ScrollView>

</ContentPage>
```

Click here to collapse or expand a tag.
This <Image> tag displays the picture of the robot in an Image control.
These two <Label> tags create Label controls that display the two lines of text.
This <Button> tag adds the Button control to the page.

The XAML for your MAUI page starts with a ContentPage tag, which can contain a single control—in this case, it's a ScrollView, which scrolls its content and displays a scrollbar on the side.

A ScrollView can contains a single control. Yours contains a VerticalStackLayout, which can contain multiple controls (like the Image, two Labels, and Button in your page) and displays them stacked on top of each other vertically.

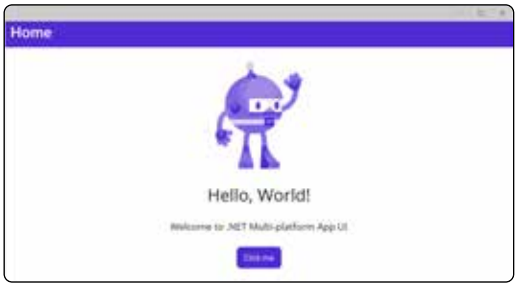
Explore your new MAUI app and figure out how it works

When you created your new .NET MAUI app, Visual Studio used a **template** to create the files for your app, substituting the name that you specified(*ExperimentWithControls*) in various lines in the files. Let's dig in to the project that you created.

Do this!

1. **Create a new .NET MAUI project called ExperimentWithControls.**
Go back to Visual Studio and **create a new .NET MAUI project**, just like you did in Chapter 1. Name your project *ExperimentWithControls*.

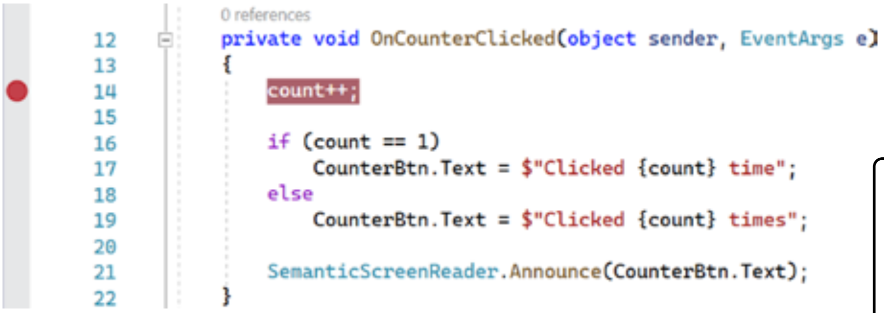
2. **Run your app and click on the button.**
When you run the app, you'll see the app window—it should look like this:



MAUI is cross-platform, which means you'll see the same app—with the same code!—whether you're running Windows or macOS.

You know you want to click the "Click me" button. Go ahead! Lhe label on the button will change from "Click me" to "Clicked 1 time" and increment (or add one) every time you click the button.

3. **Investigate how the counter on the button works.**
Go to the Solution Explorer, expand MainPage.xaml, and **open MainPage.xaml.cs**. Find the line that has the statement `count++;` and **place a breakpoint** on it.



```
0 references
private void OnCounterClicked(object sender, EventArgs e)
{
    count++;

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

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

The C# code for a page in your MAUI app is called code-behind. The XAML code and the C# code in the code-behind file work together to make your page work.

Before you go to the next step, read the code. Can you figure out how the button works?

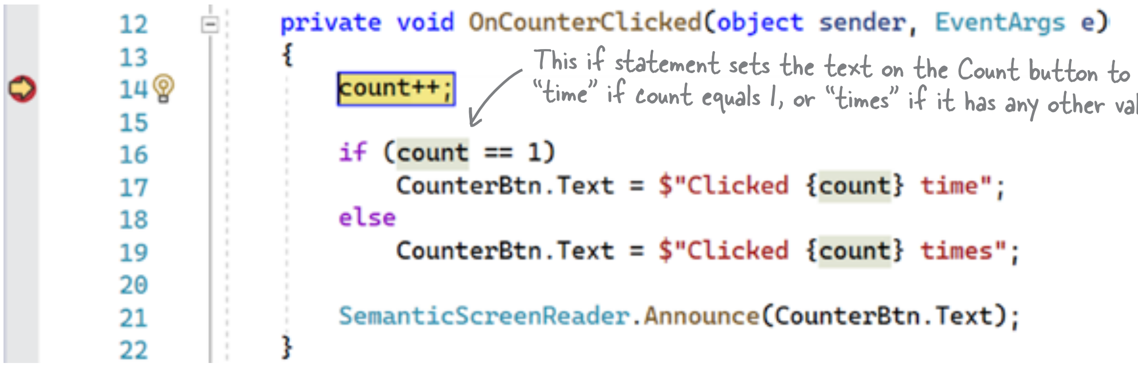
4 Click on the button and step through the code.

Add a watch for the count variable, just like you did earlier in the chapter. Then use "Step Over" to step through the code. Here's waht the OnCounterClicked event handler method does:

- First it executes `count++` to increment (or add one to) the `count` variable.
- Next it uses an if statement to check if the `count` variable equals 1. If it does, then it sets the button's text to "Clicked 1 time".
- If it doesn't equal 1, it sets the button's text to "Clicked {count} times" – you'll learn more about exactly what that the $ dollar sign and {brackets} do in Chapter 5 (it's called *string interpolation*).

5 Click on the button and step through the code.

The program should pause on the breakpoint, just like you saw earlier in the chapter:



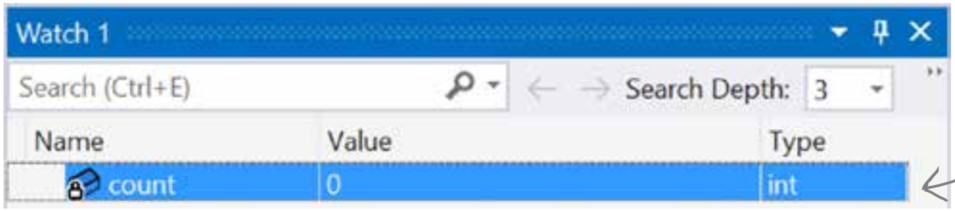
```
private void OnCounterClicked(object sender, EventArgs e)
{
    count++;

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

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

This if statement sets the text on the Count button to "time" if count equals 1, or "times" if it has any other value.

Add a watch for the `count` variable, just like you did earlier with the OperatorExamples console app. It should start out with the value 0. Step over to the



Name	Value	Type
count	0	int

The count variable starts out with the value 0. After the count++ statement is executed, the new value is 1, and every time you click the button count++ increases the value of the count variable by 1.

Keep stepping through the code. The if statement checks whether the count value is equal to 1. If it is, it executes this statement:

```
CounterBtn.Text = $"Clicked {count} time";
```

That sets the text Go back to the window with the XAML code. Find this line:

```
<Button
    x:Name="CounterBtn"
```

This is the x:Name property. It gave the button the name "CounterBtn" that you can use in code.

Every control can have a name. The `x:Name` property sets the name of the control—in this case, the button is named CounterBtn—and your C# code can use that name to make the control do things.

The XAML For Your Button Up Close

You've been editing the the XAML code in your MainPage.xaml file—are you starting to get comfortable with it? This is a great time to take a closer look at the part of your XAML that displays the button.

Here's the Button tag. Take a look at each of its five properties. Can you figure out what all do?

```
<Button
    x:Name="CounterBtn"
    Text="Click me"
    SemanticProperties.Hint="Counts the number of times you click"
    Clicked="OnCounterClicked"
    HorizontalOptions="Center" />
```

The x:Name property gives your control a name you can use in your code

The first property is x:Name, which sets the name of the control so you can use it in your C# code:

```
x:Name="CounterBtn"
```

You just saw a control name in action. When you clicked the button, the event handler method executed this statement to set the button's text, using the name CounterBtn set by the x:Name property:

```
CounterBtn.Text = $"Clicked {count} time";
```

This line uses the CounterBtn name to update the text displayed on the button.

The x:Name property gives your control a name you can use in your code

The next property determines what text is displayed n the button:

```
Text="Click me"
```

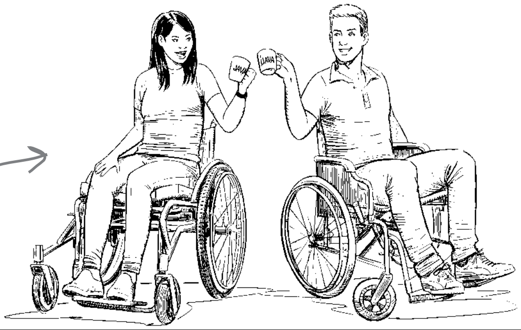
The button displays "Click me" when you first run the app. That line of code in the method changes the text to "Clicked 1 time" the first time you click it, then "Clicked 2 times" when you click it again. That line of code starts with the name of the control (CounterBtn), followed by a period, followed by Text, the name of the property.

SemanticProperties help you make your apps *accessible*

When we create our apps, we want everyone to be able to use them—and that includes people with disabilities.

A **screen reader** is a tool that lets people who are blind, visually impaired, or have learning disabilities or other conditions that interfere with their ability to read use our visual apps. Semantic properties help your app work with a screen reader.

A screen reader is an accessibility tool for people with visual, learning, or other disabilities—just like a wheelchair is an accessibility tool for people with mobility-related disabilities. They're both really important for helping to make everyday things more accessible to everyone.



The XAML For Your Button Up Close

Use a screen reader to experiment with the SemanticProperites.Hint property

The best way to make your apps accessible is to use them the way someone with accessibility issues would—in this case, using a screen reader built into your operating system.

- **In Windows, start the Narrator app.** You can run it from the Start menu, or use Windows logo key + Ctrl + Enter to turn Narrator on or off, and Windows logo key + Ctrl + N to bring up Narrator settings. Narrator will display a window with an overview of how Narrator works. It will also start to read the contents of that window, displaying a box around the section of the window that it's readingh. You can go back to that window to turn of Narrator.
- **In MacOS, start the VoiceOver utility.** It lives in the Applications/Utilities folder, but if your keyboard has Touch ID, the easiest way to turn it on or off is to press and hold the Command key while you quickly press Touch ID three times. By default the VoiceOver utility displays a welcome dialog—press the V key or click the "Use Voiceover" button to start VoiceOver.

Once you have Narrator or VoiceOver running, switch to your app window. You'll hear a voice telling you details about what's on the screen. People with visual impairments often have trouble using a mouse, so they use the keyboard to interact with apps—and you'll do the same thing. **Press the tab key** to navigate to the "Click Here" button. The screen reader will announce that you are on a button. Listen closely—you'll hear it speak the SemanticProperties.Hint value: "Counts the number of times you click"

Press Enter to click the button. Your app will execute code that includes this statement:

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

When it does, the screen reader will announce the contents of the button ("Clicked 1 time").

The Clicked property tells your app what event handler method to run when the button is clicked

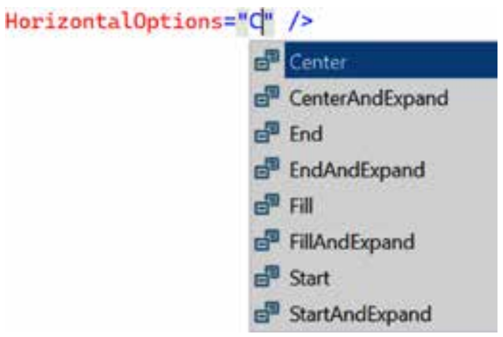
Take a look at the next property in the button's XAML code:

```
Clicked="OnCounterClicked"
```

When you clicked the button, your app used that property to figure out which method to run. You saw this in action when you placed a breakpoint on the first line of that method.

The HorizontalOptions property centers your button

When you run your app, the "Click Me" button is centered in the middle of the window. Go back to the code editor, select the word `Center` in that line of XAML code, and type C. Visual Studio will display an IntelliSense pop-up with all of the different options. Try selecting Start or End, then ren your app again—now the button will be displayed on the left or right side of the window. Experiment with all of the different horizontal options for Button control.



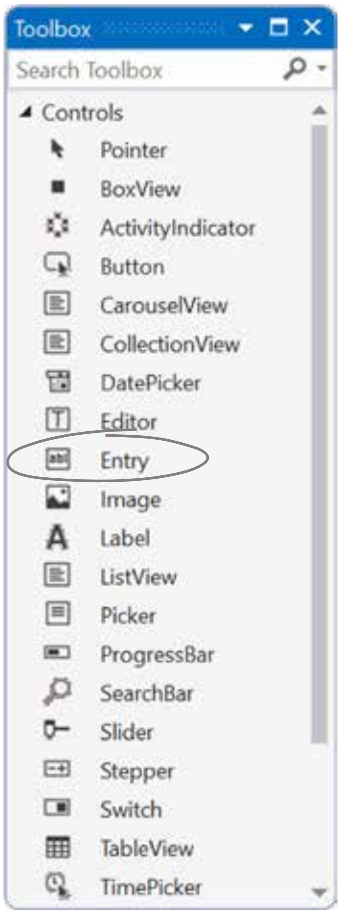
```
HorizontalOptions="C" />
```

Add an Entry control to your app

And **Entry control** gives your user a box where they can enter text. You'll add one your app—and you'll use a really useful tool in Visual Studio to do it: the **Toolbox window**. The The Toolbox is a feature of Visual Studio that makes it easy to add controls to your app.

1. Stop your app, then open the MainPage.xaml editor window in Visual Studio.
2. Place your mouse cursor just after the closing /> bracket at the end of your Button control tag, then **press enter three times** to add three blank lines. Click on the second line that you just added, so there's a blank line above your mouse cursor and another blank line below it.
3. Open the Toolbox window in Visual Studio (if it isn't already open) by **choosing "Toolbox" from the View menu**.
4. **Double-click Entry** in the Toolbox window. Visual Studio will automatically add an <Entry> tag at your cursor location, on that middle blank line you added.

Visual Studio's Toolbox window helps you add new controls to your XAML code. If you don't see the Toolbox window, choose "Toolbox" from the View menu to display it.

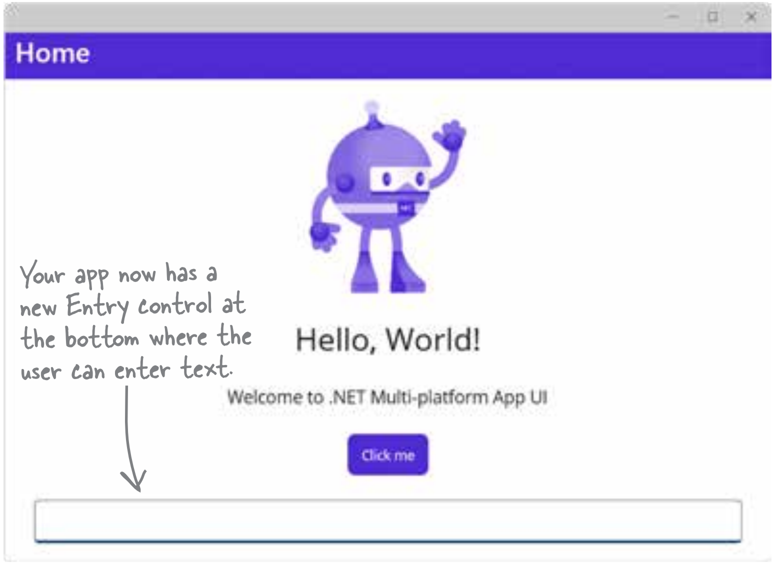


Here's what you should see in your XAML code:

```
<Button
    x:Name="CounterBtn"
    Text="Click me"
    SemanticProperties.Hint="Counts the number of times you click"
    Clicked="OnCounterClicked"
    HorizontalOptions="Center" />

<Entry Placeholder="" />
```

Now run your app. Congratulations, you just added a control for entering text!



Your app now has a new Entry control at the bottom where the user can enter text.

Add properties to your Entry control

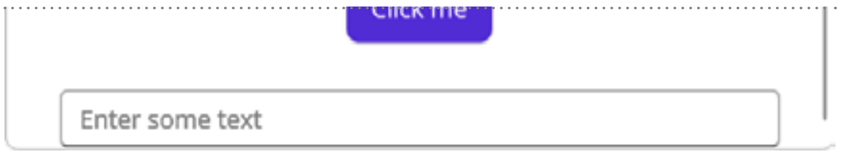
Let's make your Entry control a little more usable by adding **placeholder text**, or text that appears in a lighter color to help the user understand what they're supposed to enter.

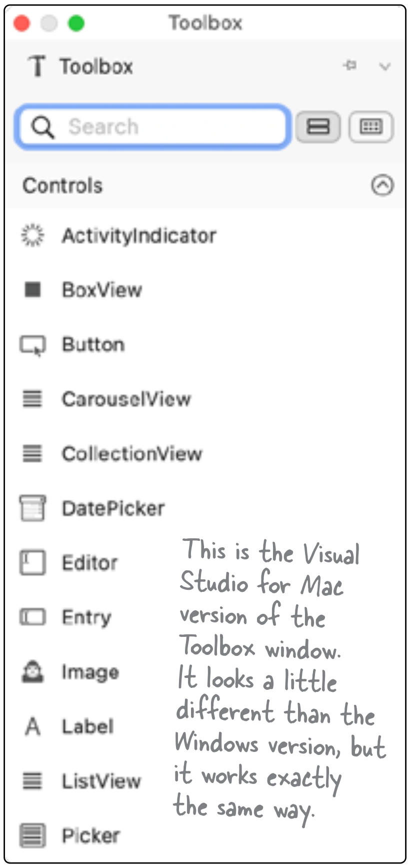
Edit the XAML code for your Entry control to **add a Placeholder property**. And since we always want our apps to be accessible to people who use screen readers, **add a SemanticProperties.Hint property** too. Notice that when you add the properties, they show up in Visual Studio's typeahead pop-up window, making it easier for you to add them.

Your Entry tag should look like this:

```
<Entry
    Placeholder="Enter some text"
    SemanticProperties.Hint="Lets you enter some text" />
```

Now run your app – you'll see a new Entry control at the bottom. The placeholder text will appear as ("Enter some text" in a lighter color, and will disappear as soon as you you type text into it.




Toolbox
Toolbox
Search
Controls
ActivityIndicator
BoxView
Button
CarouselView
CollectionView
DatePicker
Editor
Entry
Image
Label
ListView
Picker
This is the Visual Studio for Mac version of the Toolbox window. It looks a little different than the Windows version, but it works exactly the same way.

there are no Dumb Questions

Q: Why did the Entry control get added to the bottom of my app? How did it know where in the window to display?

A: When you created a new .NET MAUI app, Visual Studio used a template that generated the XAML code for the main page in the MainPage.xaml file. This file contains a set of **nested tags**, or tags that contain other tags—so one tag's start and end appear after the start and before the end of another tag. Each of these tags *creates a specific kind of control* that determines how the page is displayed.

The outermost tag in your app's XAML is a <ContentPage> opening tag, which defines a single view that contains the rest of the page. If you scroll down to the bottom of the file, you'll see the closing </ContentPage> tag. Right inside that <ContentPage> is a <ScrollView> tag—everything between the opening <ScrollView> and closing </ScrollView> tags defines contents that will automatically display a scrollbar that lets you scroll up and down if it's too long for the page. The <ScrollView> tag contains a <VerticalStackLayout> tag, with a matching </VerticalStackLayout> closing tag at the bottom. A VerticalStackLayout can contain a series of controls, one after another. Each of those controls will be displayed on the page in a vertical stack, in the order that they appear in the file.

So since the Entry control is at the bottom of the file just above the closing </VerticalStackLayout> tag, it will appear at the bottom of the page. And because it's nested inside the <ScrollView>...</ScrollView> tags, if you make your window shorter than the height of the page, you'll be able to scroll down to it.

Make your Entry control update a Label

Your app already has two Label controls. Let's add a third one and make it display everything the Entry does, so when you enter or update text in the Entry it automatically updates the Label.

① **Use the Toolbox to add a new Label control to the bottom of your page.**
When you drag the label out of the Toolbox, it will have an empty Text property:

```
<Label Text="" />
```

Change the Text property to make it display text. Then give it a SemanticProperties.Description property. This is what will get read aloud if your user is using a screen reader:

```
<Label Text="Here's what you typed:"
     SemanticProperties.Description="Here's what you typed:" />
```

You can add line breaks between properties to make them easier to read.

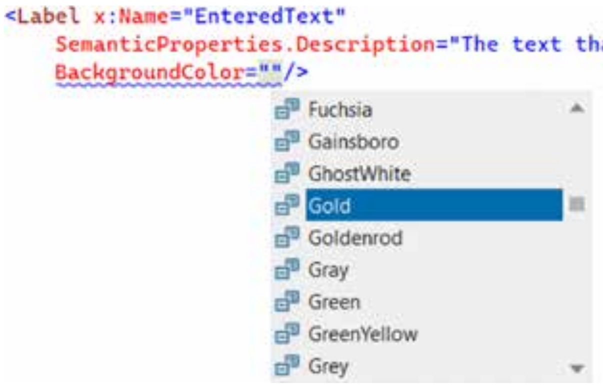
② **Use the Toolbox to add a <u>second</u> Label control under the one you just added.**
Every time the user changes the text in your Entry control, the app will update this new Label to show the text that they typed. Drag a new Label control out of the Toolbox and drop it in your XAML code between the Label control that you just added and the closing </VerticalStackLayout> tag. Then set its properties:

- ★ You'll be writing code to set the label text, so delete the Text property.
- ★ Since you're going to write code that updates the Label, you'll need to give it a name. Use an x:Name property to name it EntryOutput: `x:Name="EnteredText"`
- ★ Keep making your app accessible by adding a description for people using a screen reader: `SemanticProperties.Description="The text that the user entered"`

Your new Label should look like this:

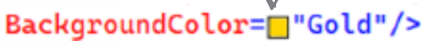
```
<Label x:Name="EnteredText"
     SemanticProperties.Description="The text that the user entered" />
```

③ **Give your label a background color.**
Add a BackgroundColor property. When you start typing, Visual Studio will pop up a IntelliSense window. Choose Gold for the background color.



```
<Label x:Name="EnteredText"
    SemanticProperties.Description="The text that the user entered"
    BackgroundColor=""/>
```

You can use Visual Studio's IntelliSense to help you add properties. Once you add it, you'll see a box with preview of the color in the XAML editor.

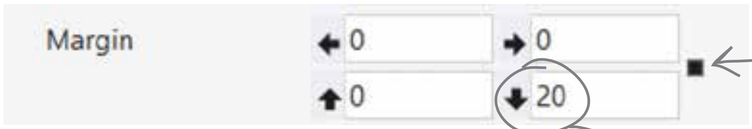


```
BackgroundColor=🟨"Gold"/>
```

4 **Use the Properties window to add a bottom margin.**

So far you've been adding properties by writing XAML code by hand. Luckily, Visual Studio has some useful tools to help you edit your XAML. The Properties window gives you an easy way to edit the properties on your controls.

Click the XAML for your Label control so the cursor is somewhere between the tags. Go to the Properties window (if you don't see it, use the View menu to display it) and find Margin. Enter 20 for the lower margin to give it a 20 pixel magin (where a pixel is 1/96th of an inch on an unscaled screen).



When this square turns black (or circle if you're using macOS), it means that the property has a value set. You can click it to see a menu.

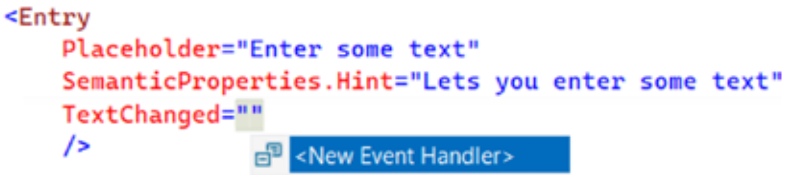
Enter 20 for the bottom margin to give 20 units of space between the label and the bottom of the page.

Your property should now look like this:

```
<Label x:Name="EnteredText"
    SemanticProperties.Description="The text that the user entered"
    BackgroundColor="Gold" Margin="0,0,0,20"/>
```

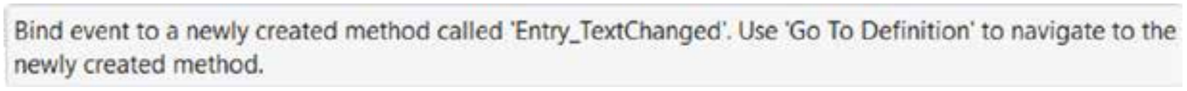
5 **Add an event handler method.**

Back in Chapter 1 you used event handler methods so your Animal Matching Game could respond to mouse clicks and timer ticks. Now you know more about C# methods—this is a good chance to apply that knowledge by creating a new event handler method to update the EnteredText control ever time the user types in the Entry control. Add a TextChanged property to your Entry control. When it comes time to enter the value, Visual Studio will suggest the value ***<New Event Handler>***:



```
<Entry
    Placeholder="Enter some text"
    SemanticProperties.Hint="Lets you enter some text"
    TextChanged=""
    />
```

Press return or use the mouse to accept the suggestion—this will cause Visual Studio to **add a new event handler method called Entry_TextChanged automatically**. You probably also noticed that it also displayed this message when you were adding the event handler:

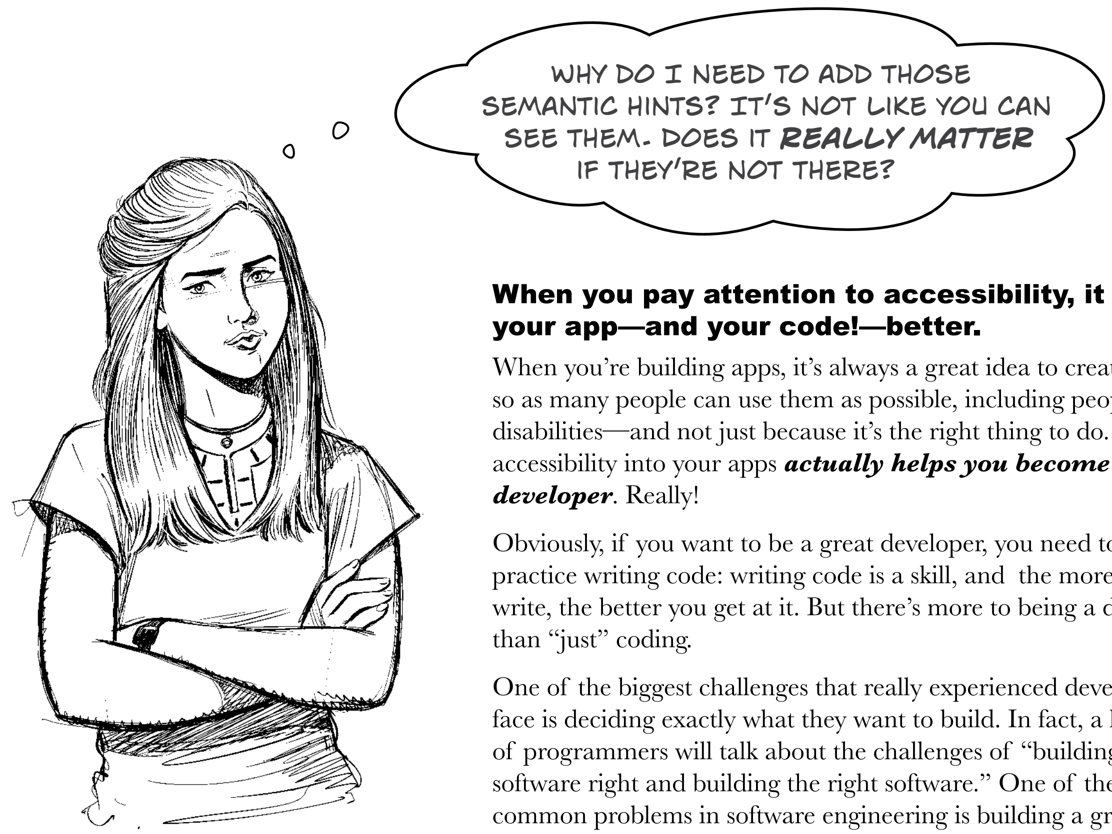


Bind event to a newly created method called 'Entry_TextChanged'. Use 'Go To Definition' to navigate to the newly created method.

Right-click on Entry_TextChanged and choose Go To Definition. This will open up MainPage.xaml.cs and jump directly to the method that Visual Studio added. Add this line of code to the method:

```
private void Entry_TextChanged(object sender, TextChangedEventArgs e)
{
     EnteredText.Text = e.NewTextValue;
}
```

Now **run your app**. You should see a label that says "Here's what you typed:" followed by a gold-colored label. Click on the Entry control and type some text—it will appear in the gold-colored label immediately.


WHY DO I NEED TO ADD THOSE SEMANTIC HINTS? IT'S NOT LIKE YOU CAN SEE THEM. DOES IT ***REALLY MATTER*** IF THEY'RE NOT THERE?

When you pay attention to accessibility, it makes your app—and your code!—better.

When you're building apps, it's always a great idea to create them so as many people can use them as possible, including people with disabilities—and not just because it's the right thing to do. Building accessibility into your apps ***actually helps you become a better developer***. Really!

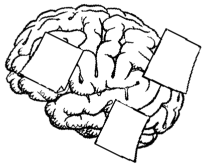
Obviously, if you want to be a great developer, you need to get practice writing code: writing code is a skill, and the more code you write, the better you get at it. But there's more to being a developer than "just" coding.

One of the biggest challenges that really experienced developers face is deciding exactly what they want to build. In fact, a lot of programmers will talk about the challenges of "building the software right and building the right software." One of the most common problems in software engineering is building a great product that doesn't do what your users need.

That's where accessibility can help you. Building accessible code well means taking the time to **really understand** how people with disabilities will use your app. Taking the time to understand and empathize with them will help you build your app better—and it's great practice for skills that will help you build the right software.

One of our big goals with this book is to help you learn important skills that will help you become an all-around great developer. Understanding your users is a really important skill, and paying attention to accessibility is a great way to get better at it!

Make it Stick

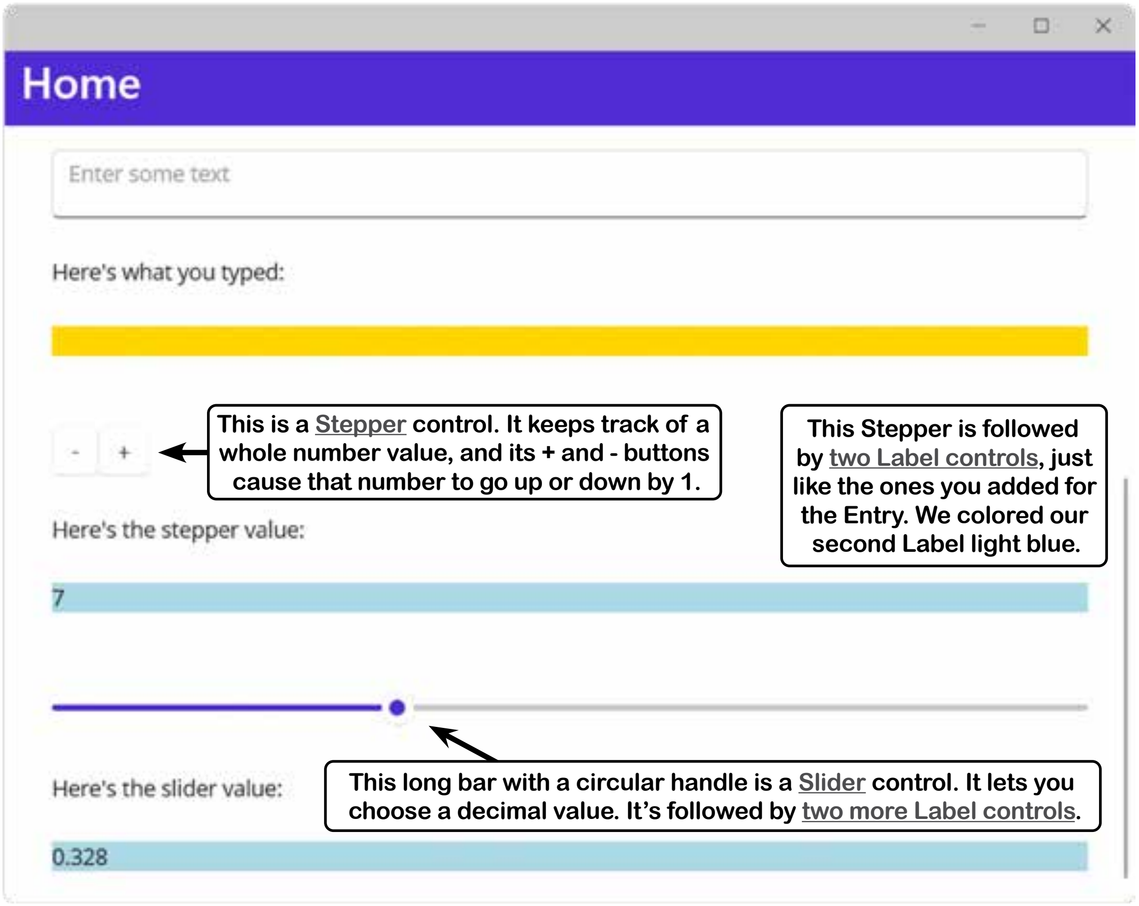


Here's an great way to get accessibility ideas to stick in your brain—especially if you don't have a disability. Turn on your screen reader, then **leave it on** while you code or do other work. Once you're used to it, **close your eyes** and keep working. Can you work using just the screen reader?

Using a screen reader is an effective way to really understandhow accessibility works.

Exercise

You added Entry and Label controls to your app—and Visual Studio's Toolbox window, Properties window, and IntelliSense helped you. Can you add six more controls to your app to let your user enter numeric values?



This is a Stepper control. It keeps track of a whole number value, and its + and - buttons cause that number to go up or down by 1.

This Stepper is followed by two Label controls, just like the ones you added for the Entry. We colored our second Label light blue.

This long bar with a circular handle is a Slider control. It lets you choose a decimal value. It's followed by two more Label controls.

Use the Toolbox window, Properties window, and Visual Studio editor to **add a Stepper control, two Label controls, a Slider control, and two more Label controls** to your app.

The two Label controls that display the values should have the BackgroundColor property set to LightBlue. **Name them StepperValue and SliderValue**. Make sure you **add SematicProperties.Description properties**.

You want your app to automatically update those the StepperValue control every time the the stepper value changes, so **add a ValueChanged event handler** to the Stepper control. Add this line of code to the event handler:

```
        StepperValue.Text = e.NewValue.ToString();
```

Then add a ValueChanged event handler **to the Slider control.** It should be identical, except that it updates the SliderValue label instead of the StepperValue label.

Don't forget to add SemanticDescription.Hint properties to your Stepper and Slider controls.

Exercise Solution

This is the Label control that was already in your XAML code — make sure you put your six new controls below it.

Here's the XAML to add the six controls to MainPage.xaml:

```
    <Label x:Name="EnteredText"
        SemanticProperties.Description="The text that the user entered"
        BackgroundColor="Gold" Margin="0,0,0,20"/>

    <Stepper Minimum="0" Maximum="10" Increment="1"
                SemanticProperties.Description="Lets you enter a whole number"
                ValueChanged="Stepper_ValueChanged" />

    <Label
        Text="Here's the stepper value:"
        SemanticProperties.Description="Here's the stepper value" />

    <Label x:Name="StepperValue"
        SemanticProperties.Description="The number the user chose with the Stepper"
        BackgroundColor="LightBlue" Margin="0,0,0,20"/>


    <Slider Minimum="0" Maximum="1" ValueChanged="Slider_ValueChanged" />

    <Label
        Text="Here's the slider value:"
        SemanticProperties.Description="Here's the stepper value" />

    <Label x:Name="SliderValue"
        SemanticProperties.Description="The number the user chose with the Slider"
        BackgroundColor="LightBlue" Margin="0,0,0,20"/>

</VerticalStackLayout>
```

These are the default properties when you drag the Stepper out of the Toolbox. Try experimenting with them.

You can add this ValueChanged property just like you did with TextChanged on your Entry control.

Here's the Slider control. It has the default properties, plus a ValueChanged property.

Here's the Label that displays the Slider value. It works exactly like the Label you used to show the value in the Entry control.

This is the closing VerticalStackLayout tag that was already in your XAML code — make sure you put your six new controls above it.

Here are the event handler methods to add to MainPage.xaml.cs:

```
private void Stepper_ValueChanged(object sender, ValueChangedEventArgs e)
{
    StepperValue.Text = e.NewValue.ToString();
}

private void Slider_ValueChanged(object sender, ValueChangedEventArgs e)
{
    SliderValue.Text = e.NewValue.ToString();
}
```

The two event handlers for the Stepper and Slider controls update the Label.

In the exercise instructions, we gave you this line of code:

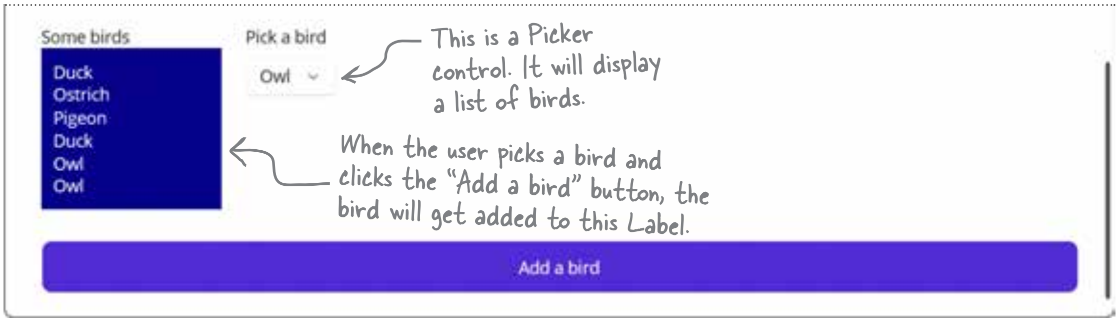
```
    SliderValue.Text = e.NewValue.ToString();
```

What do you think .ToString() does?

Here's a hint: Stepper and Slider controls can only provide numeric values, but labels can only display text.

Combine horizontal and vertical stack layouts

In this last part of the exercise you'll add a **Picker control**, which displays a list of items that you can pick from. You'll also use a Label control to display the values that were picked. Here's what it will look like:


Some birds
Duck
Ostrich
Pigeon
Duck
Owl
Owl
Pick a bird
Owl
Add a bird
This is a Picker control. It will display a list of birds.
When the user picks a bird and clicks the "Add a bird" button, the bird will get added to this Label.

Notice how the Label and Picker controls are next to each other? You'll get that layout by using a **HorizontalStackLayout control**. It works just like the VerticalStackLayout control causes all of the controls you've added to your app so far to be stacked on top of each other, except instead they get stacked next to each other.

You'll nest one Layout inside another

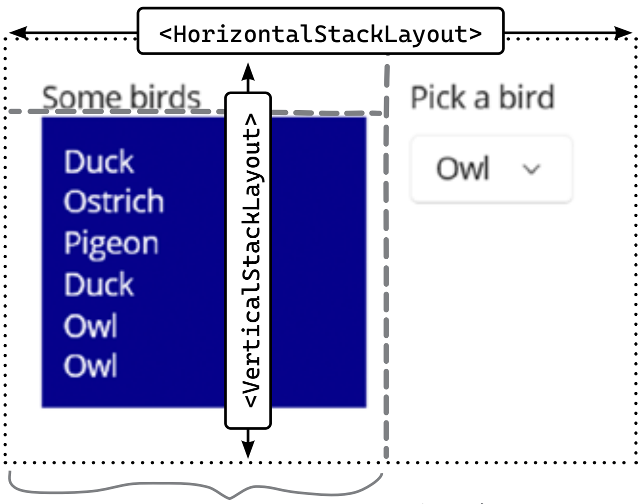
We'll use **nesting**—where one layout control lives inside another one—to create a more complex layout.

Here's how it will work:

This HorizontalStackLayout will get nested inside the outer VerticalStackLayout that's used to lay out the entire page.

```
<HorizontalStackLayout>
   <VerticalStackLayout>
      <Label Text="Some Birds" ... />
      <Label x:Name="Birds" ... />
   </VerticalStackLayout>
   <Picker x:Name="BirdPicker" ... />
</HorizontalStackLayout>
```

The HorizontalStackLayout contains two controls, a nested VerticalStackLayout and a Picker, which get stacked left to right.



This nested VerticalStackLayout contains two controls, a Label that says "Some birds" and a Label named Birds that will get updated when the user clicks the button, stacked vertically.

Add a Picker control to display a list of choices

A **Picker control** displays a list of items in a dropdown so the user can pick one of them. Let's add one to your app.

1 **Add the XAML for a Picker control and a Label for it to update.**

You've already seen how a VerticalStackLayout control lets you stack controls on top of each other. You can also stack controls horizontally by adding a **HorizontalStackLayout control**.

Go ahead and **add this XAML code** just above the closing </VerticalStackLayout> tag. You can type it all or use the Toolbox. When you add Clicked event for the button, **press Tab** to let Visual Studio generate an event handler method for you, just like you did earlier..

```
    <HorizontalStackLayout Spacing="20">

        <Picker x:Name="BirdPicker" Title="Pick a bird" />

        <VerticalStackLayout>

            <Label Text="Some birds" SemanticProperties.Description="Some birds"/>

            <Label x:Name="Birds"
                    Padding="10" MinimumWidthRequest="150"
                    TextColor="White" BackgroundColor="DarkBlue"
                    SemanticProperties.Description="Shows the added birds" />

        </VerticalStackLayout>

    </HorizontalStackLayout>

    <Button x:Name="AddBird" Clicked="AddBird_Clicked" Text="Add a bird"
            Margin="0,0,0,20" SemanticProperties.Hint="Adds a bird"/>

</VerticalStackLayou>
```

This <Button ... /> tag should be just above the closing </VerticalStackLayout> tag that's already in the MainPage.xaml file.

2 **Initialize the Picker with a list of birds.**

Open the MainPage.xaml.cs file and find the MainPage method at the top. This method gets run every time the page loads. Insert two lines after `InitializeComponent();` and **add this code**.

```
public MainPage()
{
    InitializeComponent();

    BirdPicker.ItemsSource = new string[] {
        "Duck",
        "Pigeon",
        "Penguin",
        "Ostrich",
        "Owl"
    };
}
```

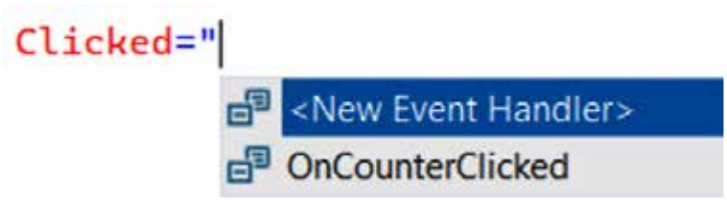
You used the x:Name property to name your Picker control "BirdPicker" – this sets the list of items in the picker that get displayed when the user clicks on it.

Put your code at the end of the MainPage method.

Open the MainPage.xaml.cs file and add this code to the MainPage method. Be careful with the square and curly braces, quotes, and commas.

3 Fill in the event handler for the Button control.

When you added the XAML for the Button control, you used Visual Studio's IntellliSense pop-up to help you add a new event handler to the C# code:



```
Clicked="|
```

Since you used the x:Name property to name your Picker control AddBird, Visual Studio created an empty event handler method called AddBird_Clicked:

```
private void AddBird_Clicked(object sender, EventArgs e)
{


}
```

Add this line of code to the AddBird_Clicked method:

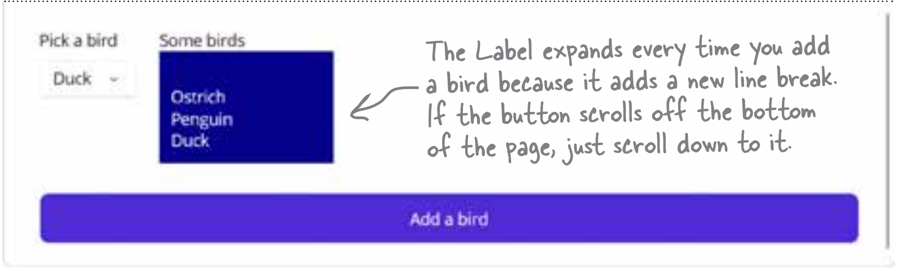
```
private void AddBird_Clicked(object sender, EventArgs e)
{
    Birds.Text = Birds.Text + Environment.NewLine + BirdPicker.SelectedItem;
}
```

Take a closer look at the line of code—let's break down exactly what it does.

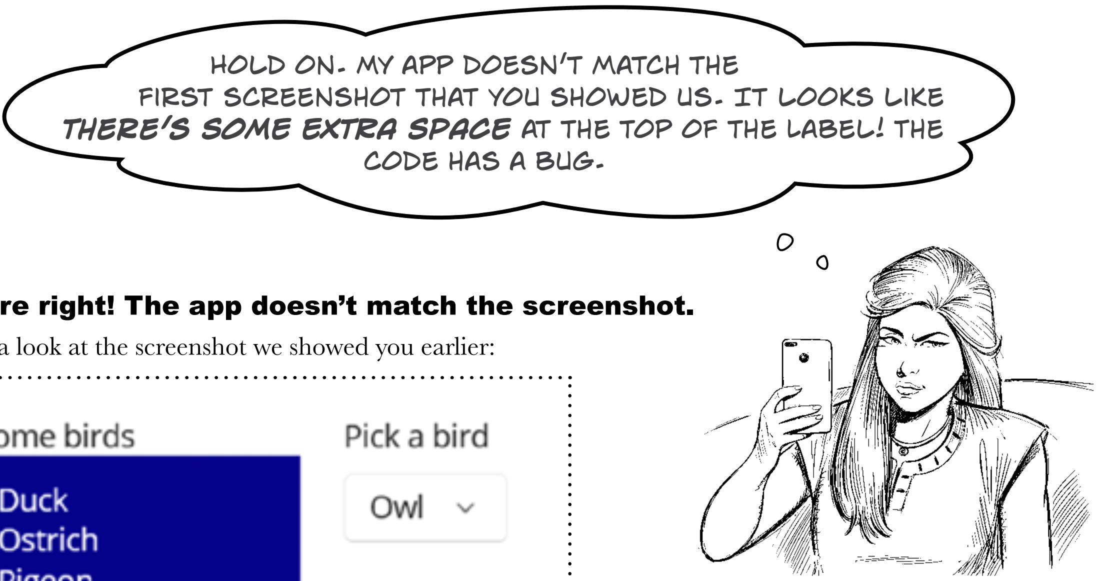
1. The line starts with `Birds.Text =` ... which means it's setting the text in the Bird label.
2. The text is being set to `Birds.Text +` followed by additional things—this means it's going to take whatever is in the Label and **append text** to it, or add additional text to the end.
3. The first thing that gets appended is `Environment.NewLine`, which adds a line break. The Label control will display **multi-line text**, adding a line break every time it sees a line break.
4. After the line break, it appends `BirdPicker.SelectedItem`—this is the item that's currently selected in the Picker control.

4 Run your app and use your new Picker control.

Scroll to the bottom of the app, choose a bird from the Picker, and click the Add a bird button—it will get added to the Label that contains the birds. Select a few more birds and add them.

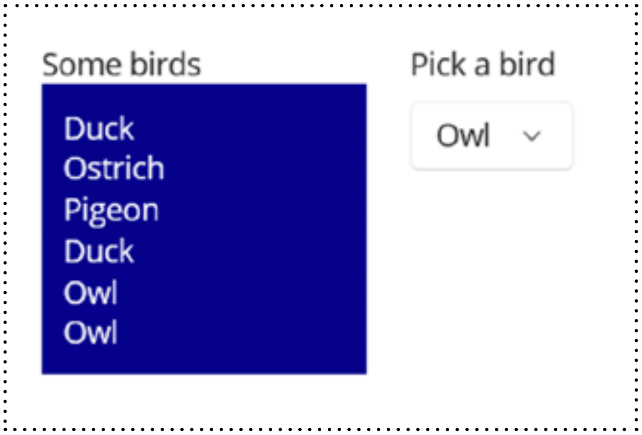


Pick a bird
Duck
Some birds
Ostrich
Penguin
Duck
Add a bird
The Label expands every time you add a bird because it adds a new line break. If the button scrolls off the bottom of the page, just scroll down to it.

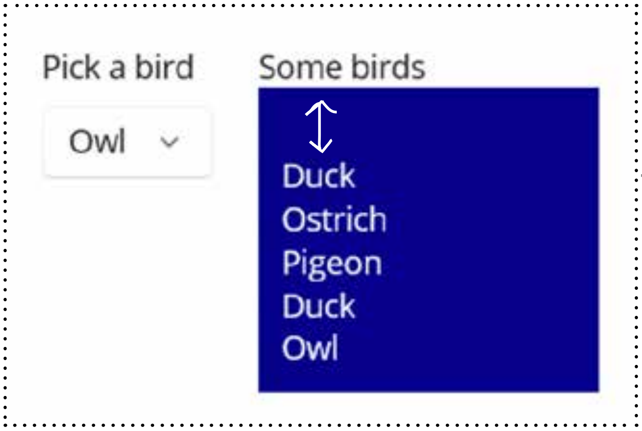

HOLD ON. MY APP DOESN'T MATCH THE FIRST SCREENSHOT THAT YOU SHOWED US. IT LOOKS LIKE ***THERE'S SOME EXTRA SPACE*** AT THE TOP OF THE LABEL! THE CODE HAS A BUG.

You're right! The app doesn't match the screenshot.

Take a look at the screenshot we showed you earlier:



Run your app and try adding those same birds. When you get to the first owl, you'll see extra space at the top of the Label:



Oops! It looks like we've got some extra space at the top of the Label that shows the birds that you picked.

Looks like we've got a bug. Time to put on your Sherlock Holmes cap.
Let's sleuth out this bug!

Sleuth it Out

The Case of the Extraneous Space

Understanding a bug is the first step in fixing it.

In the last chapter, we looked at the code carefully and found several clues to help us solve the Case of the Unexpected Match. But as you keep going through this book, your apps will get longer and longer, and while looking at the code is a good start, it may not always be the best way to figure out what's causing a bug.

Luckily, the Visual Studio debugger is a great tool for that. (That's why it's called a debugger: it's a tool that helps you get rid of bugs!)

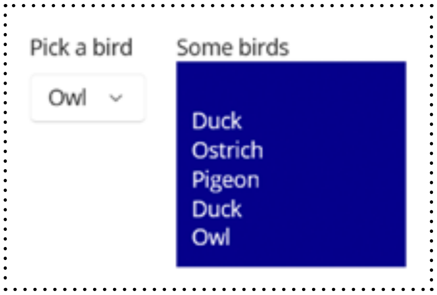
Reproduce the bug

It's obvious that there's a problem! But as Sherlock Holmes once said, "There is nothing more deceptive than an obvious fact." When you're sleuthing out bugs, can't just rely on what's obvious. You need to confirm for yourself exactly what's going on. The way to do that is to **reproduce the bug**.

Stop your app. Make sure it's not running, so you've got a fresh start. Then do this:

1. Start your app again
2. Pick Duck and click the "Add a bird" button
3. Pick Ostrich and click the "Add a bird" button
4. Pick Pigeon and click the "Add a bird" button
5. Pick Duck and click the "Add a bird" button
6. Pick Owl and click the "Add a bird" button

Your app should now look exactly like the screenshot:



```

Duck
Ostrich
Pigeon
Duck
Owl
```

"There is nothing more deceptive than an obvious fact."
– Sherlock Holmes

Now restart your app, then try it again with different birds. You should still see extra space at the top of the Label. You can make the bug happen over and over again, at will. That means the problem is **reproducible**: you can follow a set of stpes to make it happen. Reproducing a bug is a great first step to fixing it.

Before you go on, can you sleuth out what's causing the extra space to get added?

Sleuth it Out

Every good investigation starts by identifying a list of suspects

When you're tracking down a bug, what's the first thing you should do? You could start placing breakpoints in the code… but where? **The first step in debugging is thinking.** Look at your code, think about how it works, and try to imagine where the bug might be. That will help you figure out where to put your breakpoints.

So let's think through the code. It starts with a button—and the button calls a method:

```
<Button x:Name="AddBird" Clicked="AddBird_Clicked" Text="Add a bird"
        Margin="0,0,0,20" SemanticProperties.Hint="Adds a bird"/>
```

All of the code to add the bird to the Label is in that **AddBird_Clicked method**. Now we have a suspect!

IDE Tip: Using the debugger

You're going to be using the debugger a lot in this book! We've walked you through it a few times, but you're as you get further in the book and write more and more code, you should feel comfortable using the debugger on your own.

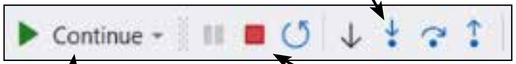
Let's start with **a few tips** to help you get comfortable debugging your code.

- ★ Think before you debug. Read through your code. Understand how it works (and not just how *you think it works*).
- ★ Use the Watch window, Locals window, and hovering over variables to keep track of their values. They all do the same thing—show you the value of a variable—so you can decide which one you feel most comfortable with.
- ★ Don't be afraid to restart your app. Stop and start your code frequently—every time you run your code, you're *running an experiment*. Run it as many times as it takes to understand what's going on.

Here's a handly **list of the debugger commands**. They may feel strange at first, but they'll be second nature soon.

- ★ When you press the triangle Run button in the toolbar or choose Start Debugging (F5 or ⌘↩), Visual Studio starts running your code in the debugger. You can place a breakpoint whether or not the debugger is running.
- ★ To place a breakpoint, click on a line of code and choose Toggle Breakpoint (F9 or ⌘/) form the Debug menu.
- ★ When your code hits a breakpoint, it stops running so you can inspect variables.
- ★ When Visual Studio breaks on a breakpoint, the toolbar shows you the commands you can use to keep executing. Debugging code can be a little weird to get used to if you haven't done it before, so try sticking to just these four commands—here's where you'll find them in the toolbar (for Windows or macOS) and using keyboard shortcuts:

Step Over (F10) executes the current statement and breaks on the next one.

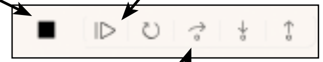


Continue Debugging (F5) starts the app running again.

Stop Debugging (Shift F5) stops the debugger.

Stop Debugging (⇧⌘↩) stops the debugger.

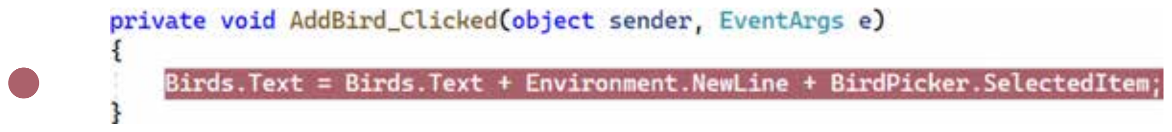
Continue Debugging (⌘↩) starts the app running again.



Step Over (⇧⌘O) executes the current statement and breaks on the next one.

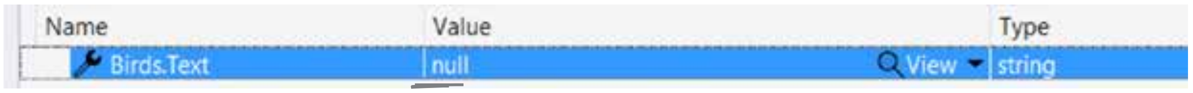
Add a breakpoint and start debugging the code

Now that we have a suspect, let's catch it in the act. Add a breakpoint to the line in the AddBird_Clicked method:



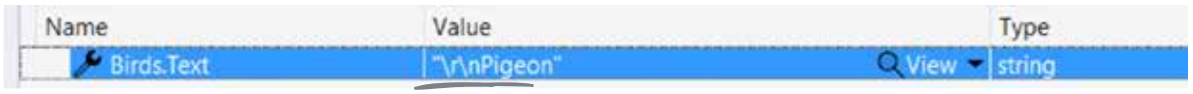
```
private void AddBird_Clicked(object sender, EventArgs e)
{
    Birds.Text = Birds.Text + Environment.NewLine + BirdPicker.SelectedItem;
}
```

Now **run your code**. Pick a bird, then click the "Add a bird" button. The debugger stops on your breakpoint. Next, **add a watch for Birds.Text**, just like you did earlier in the chapter. The value should be **`null`**.



Name	Value	Type
Birds.Text	null	string

Then **step over that line of code** (F10 on Windows or ⇧⌘O on macOS) to run it. You should see this value:

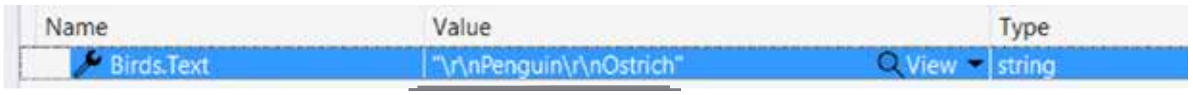


Name	Value	Type
Birds.Text	"\r\nPigeon"	string

The value of Birds.Text is a string: `\r\n` followed by the bird you picked. What do you think `\r\n` does?

NOTE: If you're using macOS, you'll see `\n` instead of `\r\n`.

Continue debugging (F5 or ⌘↩) to start your app running again. Pick a different bird and step over the line of code. Now have a look at the Birds.Text watch:



Name	Value	Type
Birds.Text	"\r\nPenguin\r\nOstrich"	string

Repeat the process a few more times: continue debugging, pick a bird, click the button, step over, check the watch. Eventually your Birds.Text value will look something like this (you'll see `\n` instead of `\r\n` on macOS):

```
"\r\nPenguin\r\nOstrich\r\nPigeon\r\nDuck\r\nOwl\r\nPigeon\r\nDuck\r\nOwl"
```

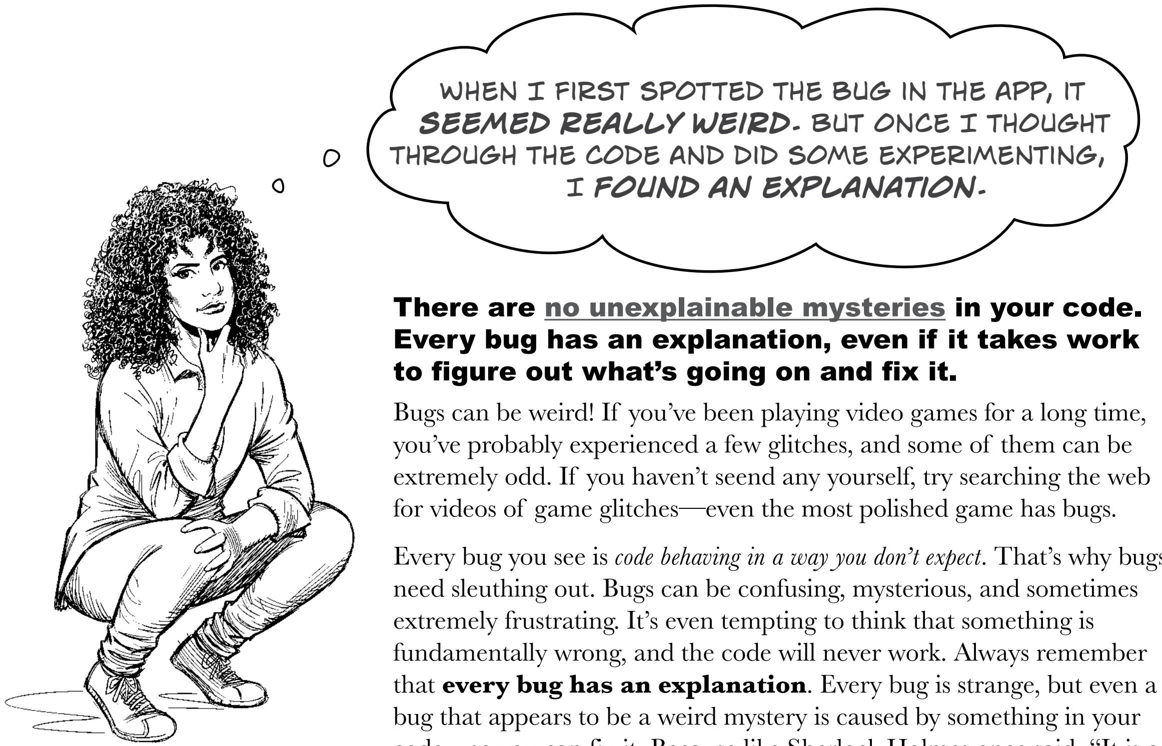
You've probably figured out by now that **the \r\n or \n is the line break**. The first time the AddBird_Clicked method is called, the Label text is empty (that's what the `null` value means), so when the app adds the current value (empty) plus a line break plus the bird, it adds an extra line break to at the start of the string.

Now that we've found the culprit, we can fix the app. **Replace the AddBird_Clicked** method with this code, which uses a special method, **String.IsNullOrEmpty**, which checks if a string is empty:

```
private void AddBird_Clicked(object sender, EventArgs e)
{
    if (!String.IsNullOrEmpty(Birds.Text))
    {
        Birds.Text = Birds.Text + Environment.NewLine;
    }
    Birds.Text += BirdPicker.SelectedItem;
}
```

String.IsNullOrEmpty(Birds.Text) checks the value of Birds.Text and returns true if it's empty or false if it's not. The ! in front of it reverses that value, so the line break is only added if Birds.Text is empty. You'll learn all about how a method can return a value in the next chapter.

Run your app again and add a few birds to the Label—there's no more empty space above it. **Your app is fixed!**


WHEN I FIRST SPOTTED THE BUG IN THE APP, IT SEEMED REALLY WEIRD. BUT ONCE I THOUGHT THROUGH THE CODE AND DID SOME EXPERIMENTING, I FOUND AN EXPLANATION.

There are no unexplainable mysteries in your code. Every bug has an explanation, even if it takes work to figure out what's going on and fix it.

Bugs can be weird! If you've been playing video games for a long time, you've probably experienced a few glitches, and some of them can be extremely odd. If you haven't seend any yourself, try searching the web for videos of game glitches—even the most polished game has bugs.

Every bug you see is *code behaving in a way you don't expect*. That's why bugs need sleuthing out. Bugs can be confusing, mysterious, and sometimes extremely frustrating. It's even tempting to think that something is fundamentally wrong, and the code will never work. Always remember that **every bug has an explanation**. Every bug is strange, but even a bug that appears to be a weird mystery is caused by something in your code—so you can fix it. Because like Sherlock Holmes once said, "It is a mistake to confound strangeness with mystery."

Bullet Points

- You'll use many different **controls** to build your app's user interface (or UI). The UI is the part of the application that your user interacts with.
- The C# code for a page in a MAUI app is called **code-behind**. The XAML code and the C# code in the code-behind file work together to make the page work.
- The x:Name property gives your control a name you can use in your code.
- When you pay attention to **accessibility**, it makes your app—and your code!—better. **Semantic properties** help you make your apps accessible by providing descriptions and hints for people who use screen readers.
- In XAML you can have **nested controls**, or tags that contain other controls, so one control's start and end tag appear after the start tag and before the end of another tag.
- You can use nested **HorizontalStackLayout** and **VerticalStackLayout** controls to create more complex layouts.
- The first step in debugging is thinking: look at your code, think about how it works, and try to imagine where the bug might be.
- **Reproducing a bug** is an important tool that helps you fix it. When you're debugging, you're **running an experiment** every time you run your code. Run it as many times as it takes to understand what's going on.