O'REILLY®

~~Fourth~~ Fifth Edition

Head First C#

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.

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

Andrew Stellman
& Jennifer Greene

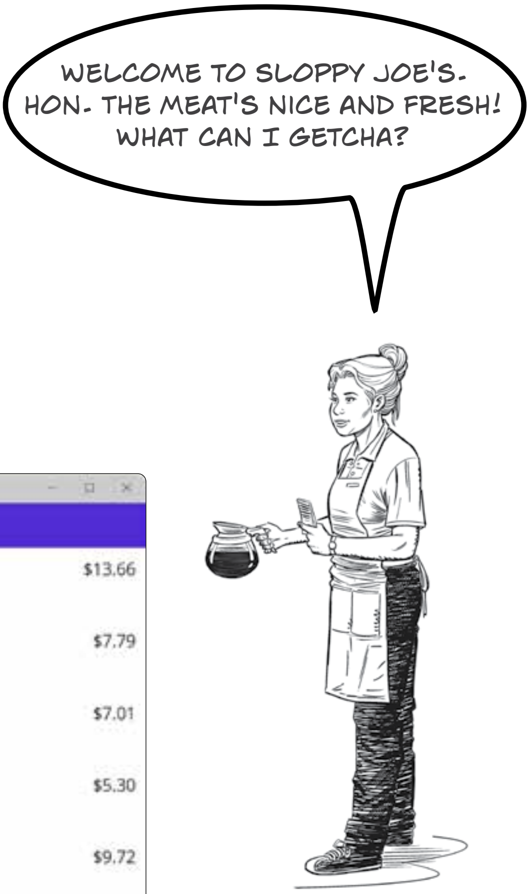
This is the .NET MAUI project from Chapter 4. You'll learn about Grid controls, then build an app that generates a menu for Sloppy Joe's diner with random prices.

A Brain-Friendly Guide

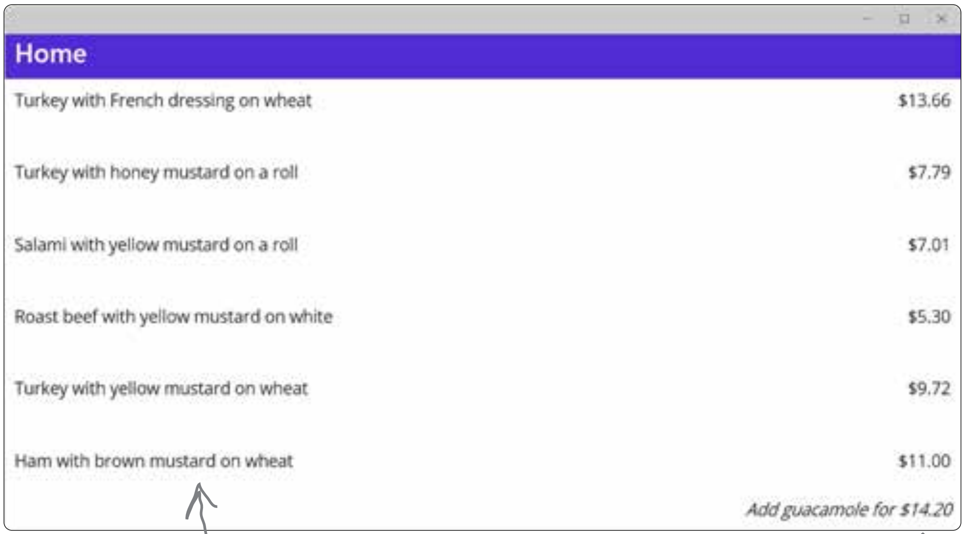
Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!

Sloppy Joe has a pile of meat, a whole lotta bread, and more condiments than you can shake a stick at. What he doesn't have is a menu! Can you build a program that makes a new random menu for him every day? You definitely can…with a **new MAUI app**, some arrays, your handy random number generator, and a couple of new, useful tools. Let's get started!

Here's the app you'll build. It creates a menu with six random sandwiches. Each sandwich has a protein, a condiment, and a bread, all chosen at random from a list. Every sandwich is gets a random price, and there's a special random price at the bottom to add guacamole on the side.


WELCOME TO SLOPPY JOE'S. HON. THE MEAT'S NICE AND FRESH! WHAT CAN I GETCHA?

Sloppy Joe needs a new menu every day. Your app will generate random sandwiches and prices for him.



```
Home
Turkey with French dressing on wheat          $13.66
Turkey with honey mustard on a roll            $7.79
Salami with yellow mustard on a roll           $7.01
Roast beef with yellow mustard on white        $5.30
Turkey with yellow mustard on wheat            $9.72
Ham with brown mustard on wheat               $11.00
                          Add guacamole for $14.20
```

Each sandwich is generated by choosing a random protein, random condiment, and random bread from arrays.

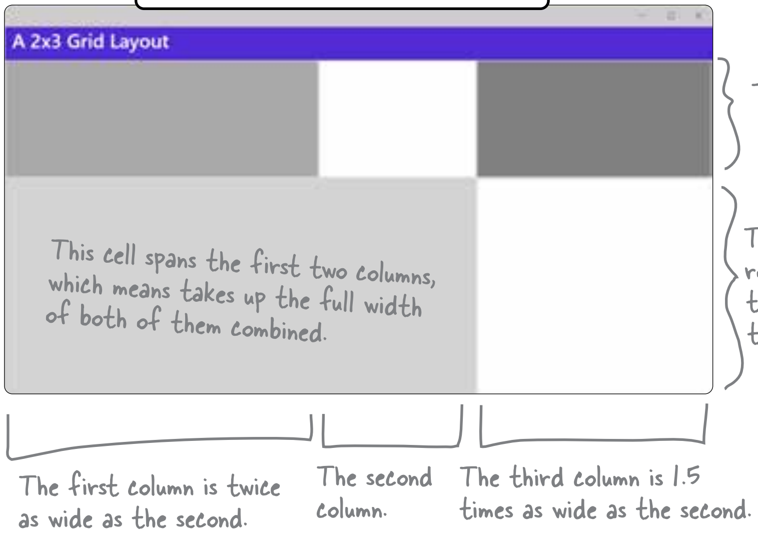
The prices are random numbers between 5.00 and 14.99.

Sloppy Joe's menu app uses a Grid layout

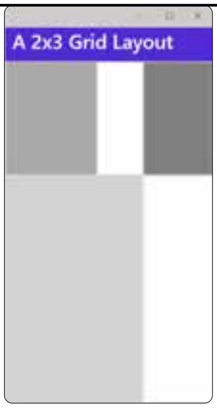
A **Grid control** contains other controls, and defines a set of rows and columns to lay out those controls.

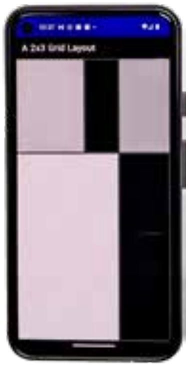
You've used other layout controls: you've used VerticalStackLayout controls to stack Button, Label, and other controls in your apps on top of each other. You used a HorizontalStackLayout control in Chapter 2 for your bird picker. And in the Animal Matching Game project your VerticalStackLayout contained a FlexLayout that arranged the buttons so they stacked horizontally, flowing into rows if as the window size changed.

Here's an example of a Grid layout with two rows and three columns.


A 2x3 Grid Layout
This is a row.
The second row height is twice as tall as the first row.
This cell spans the first two columns, which means takes up the full width of both of them combined.
The first column is twice as wide as the second.
The second column.
The third column is 1.5 times as wide as the second.

The layout adjusts as you change the window size.





The Grid preserves the row and column proportions when you change the size and shape of the page, which is really useful when you want your app to run on devices with different screen sizes.

Watch it! **A Grid control is for <u>layouts</u>, not data.**

When most of us see something that contains "rows" and "columns" we think of tables of data, like spreadsheets or HTML tables. That's not what a Grid control is all about.

*The Grid control is for **laying out content**. Its job is to contain other controls, and give you a way to design more interesting or intricate layouts than you get with stack panels, in a way that works well with different window sizes or on mobile devices.*

Grid controls

The Grid control contains other controls, and works just like the other layout controls to contain **child controls** (the other controls nested inside it). There's an opening <Grid> tag and a closing </Grid> tag, and the tags for all of the child controls are between them.

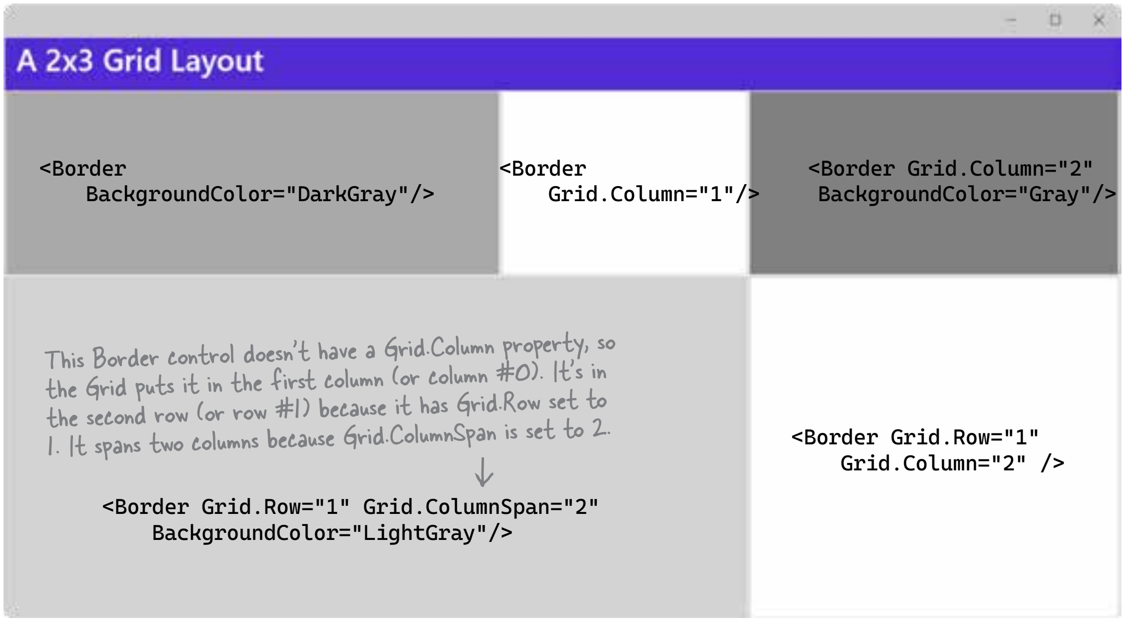
Cells in a grid are invisible—their only purpose is to determine where the child controls are displayed on the page. We used **Border controls** to make the grid visible. A Border control draws a border around a child control nested inside it:

```
<Border>
    <Label Text="I have a border!"/>
</Border>
```

A Border can only contain one child control. In the app below we didn't nest any controls inside the Borders—we just took advantage of the fact that each Border fills up the entire cell. We used the Border control's BackgroundColor property to make some of the cells in the grid darker.

Use Grid properties to put a control in a cell

The rows sand columns in a Grid are numbered starting with 0. To put a child control in a specific row and column, use the Grid.Row and Grid.Column properties. For example, putting `<Border Grid.Row="1" Grid.Column="2" />` between Grid tags will make the Grid place the border in the second row and third column. You can also make a control span multiple rows or columns using the Grid.RowSpan and Grid.ColumnSpan properties.



```
<Border
    BackgroundColor="DarkGray"/>
```

```
<Border
    Grid.Column="1"/>
```

```
<Border Grid.Column="2"
 BackgroundColor="Gray"/>
```

This Border control doesn't have a Grid.Column property, so the Grid puts it in the first column (or column #0). It's in the second row (or row #1) because it has Grid.Row set to 1. It spans two columns because Grid.ColumnSpan is set to 2.

```
<Border Grid.Row="1" Grid.ColumnSpan="2"
    BackgroundColor="LightGray"/>
```

```
<Border Grid.Row="1"
    Grid.Column="2" />
```

Define the rows and columns for a Grid

The Grid control XAML has sections to define rows and columns. Each row or column can either have proportional sizes—for example, column 3 is twice as wide as column 2 and three times as wide as column 1—or absolute sizes in device-independent pixels.

The row and column definitions are in special sections inside the <Grid> tag. The row definitions are inside a <Grid.RowDefinitions> section, and the column definitions are inside a <Grid.ColumnDefinitions> section.

Here's the complete XAML for the app that we've been showing you. **Create a .NET MAUI app** called *GridExample* and add this XAML code (and delete the OnCounterClicked method in MainPage.xaml.cs).

Do this!

```
<?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="GridExample.MainPage">

    <ScrollView>

        <Grid>

            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition Height="2*"/>
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="2*"/>
                <ColumnDefinition/>
                <ColumnDefinition Width="1.5*"/>
            </Grid.ColumnDefinitions>

            <Border BackgroundColor="DarkGray"/>
            <Border Grid.Column="1" />
            <Border Grid.Column="2" BackgroundColor="Gray"/>

            <Border Grid.Row="1" Grid.ColumnSpan="2" BackgroundColor="LightGray"/>
            <Border Grid.Row="1" Grid.Column="2" />

        </Grid>

    </ScrollView>

</ContentPage>
```

The app has two rows, so the Grid.RowDefinitions section contains two RowDefinition tags. The second row height is twice as tall as the first row, so we added the `Height="2*"` property to the second RowDefinition tag to make it twice as tall.

The Grid.ColumnDefinitions section has three ColumnDefinition tags, one for each of the three columns. The first column is twice as wide as the second, so it has `Width="2*"`. The third column is 1.5 times as wide, so it has `Width="1.5*"`.

Here are the child Border controls we just showed you.

Row heights and column widths

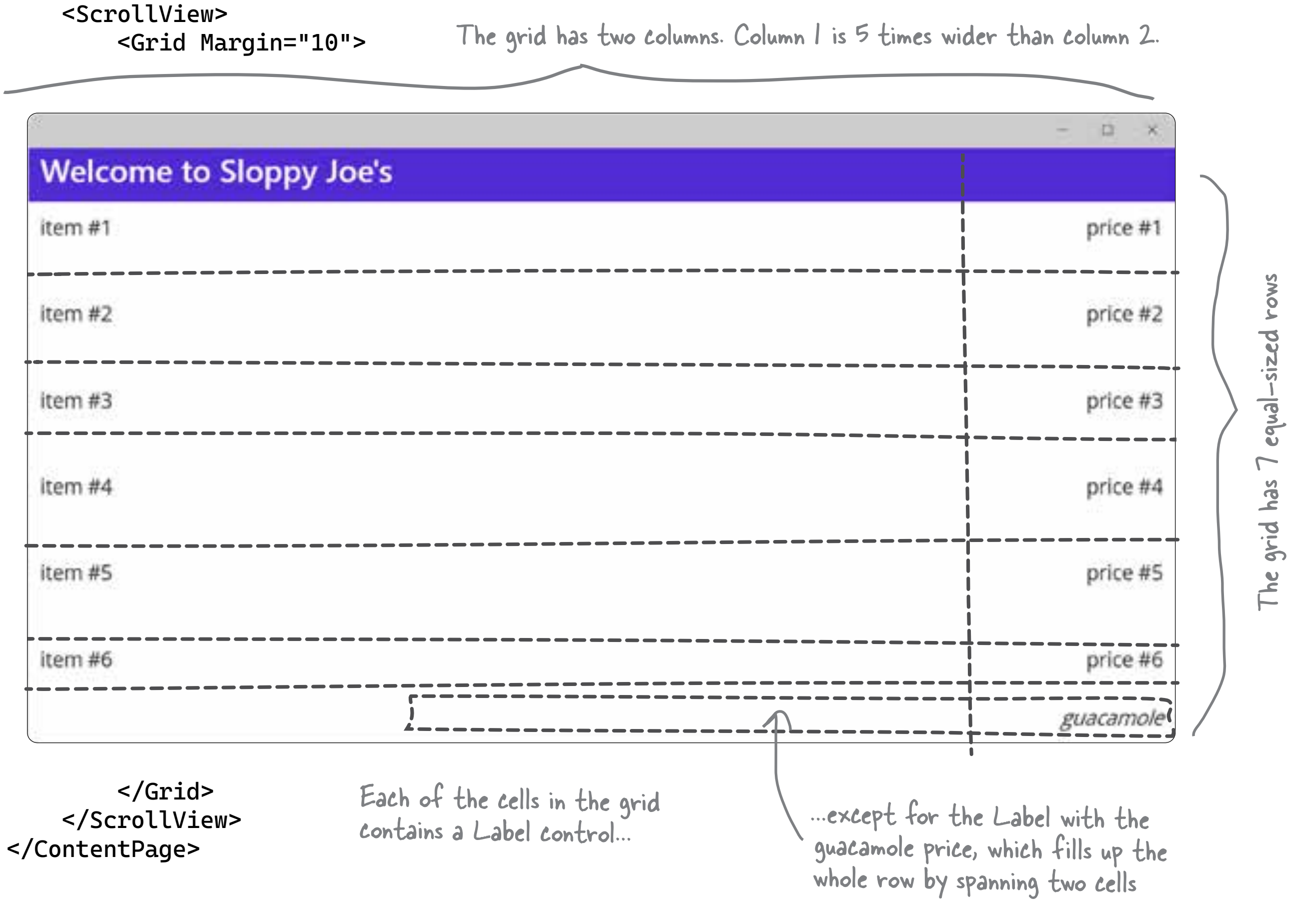
When you use a value like 2* in a RowDefinition.Height or ColumnDefinition.Width property, you're choosing a proportional width, which means they're proportional to each other. You'll get the same results setting the first row to 6* and the second row to 12* because the proportions are still the same: the second row is still twice as big as the first row.

You can also set a row width or column height to an absolute value like 100, which will cause it to be sized in device-independent pixels. If all the rows and columns are proportional, the grid will fill up the page. If you set an absolute width or height, it could end up larger than the page, which is why it's a good idea to nest the Grid inside a ScrollView.

Create the Sloppy Joe's menu app and set up the grid

Create a new .NET MAUI app and name it SloppyJoe. The first thing you'll do is create the XAML for the app. Here's how it will work:

```
<ContentPage>
    <ScrollView>
        <Grid Margin="10">
```


The grid has two columns. Column 1 is 5 times wider than column 2.
Welcome to Sloppy Joe's
item #1
price #1
item #2
price #2
item #3
price #3
item #4
price #4
item #5
price #5
item #6
price #6
guacamole
The grid has 7 equal-sized rows
Each of the cells in the grid contains a Label control...
...except for the Label with the guacamole price, which fills up the whole row by spanning two cells

```
        </Grid>
    </ScrollView>
</ContentPage>
```

We'll give you all of the XAML for the app. But before we do, try editing the MainPage.xaml file and creating the XAML for the page on your own. Can you use the app we just gave you as an example to create the row and column definitions yourself?

See how far you can get, then comapre it with our XAML.

Here's the XAML for the app.

Take your time and go through it line by line to make sure you understand how its grid works.

```
<?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="SloppyJoe.MainPage">

    <ScrollView>

        <Grid Margin="10">
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="5*"/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>

            <Label x:Name="item1" FontSize="18" Text="item #1" />
            <Label x:Name="price1" FontSize="18" HorizontalOptions="End"
                   Grid.Column="1" Text="price #1"/>

            <Label x:Name="item2" FontSize="18" Grid.Row="1" Text="item #2"/>
            <Label x:Name="price2" FontSize="18" HorizontalOptions="End"
                   Grid.Row="1" Grid.Column="1" Text="price #2"/>

            <Label x:Name="item3" FontSize="18" Grid.Row="2" Text="item #3" />
            <Label x:Name="price3" FontSize="18" HorizontalOptions="End"
                   Grid.Row="2" Grid.Column="1" Text="price #3"/>

            <Label x:Name="item4" FontSize="18" Grid.Row="3" Text="item #4" />
            <Label x:Name="price4" FontSize="18" HorizontalOptions="End"
                   Grid.Row="3" Grid.Column="1" Text="price #4"/>

            <Label x:Name="item5" FontSize="18" Grid.Row="4" Text="item #5" />
            <Label x:Name="price5" FontSize="18" HorizontalOptions="End"
                   Grid.Row="4" Grid.Column="1" Text="price #5"/>

            <Label x:Name="item6" FontSize="18" Grid.Row="5" Text="item #6" />
            <Label x:Name="price6" FontSize="18" HorizontalOptions="End"
                   Grid.Row="5" Grid.Column="1" Text="price #6"/>

            <Label x:Name="guacamole" FontSize="18" FontAttributes="Italic" Text="guacamole"
                   Grid.Row="6" Grid.ColumnSpan="2" HorizontalOptions="End" VerticalOptions="End" />

        </Grid>
    </ScrollView>

</ContentPage>
```

If you used a different app name, you'll see a different namespace here.

The 10 pixel margin around the grid adds a little space between the Labels and the edge of the window.

The grid has six rows that are all the same height.

After you add this code to your MainPage.xaml file, don't foget to go to the MainPage.xaml.cs file and delete the OnCounterClicked method and count field.

The grid has two columns. The first column is five times wider than the second.

Each of these Label controls goes in a different cell. We gave each of them text like "item #1" or "price #3" to make it easier to see how the grid is laid out when you run the app.

Each price has its HorizontalOptions set to "End" so it gets aligned all the way to the right of the window.

This Label spans both columns in the bottom row, so it stretches across two cells. Try removing the Grid.ColumnSpan property—what happens?

The Label with the guacamole price has both its Horizontal and Vertical options set to "End" to align it to the bottom right corner of the cell.

The C# code for the main page

Here's the C# code for the main page of your Sloppy Joe app. We're about to give you an exercise to build a class called MenuItem that generates random sandwiches and prices. As soon as the page loads, it calls a method called MakeTheMenu that uses an array of MenuItem objects to fill in all of the prices, and one last MenuItem object to get the price for the guacamole.

Your namespace will be different if you chose a different name for your app.

```
namespace SloppyJoe;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        MakeTheMenu();
    }

    private void MakeTheMenu()
    {
        MenuItem[] menuItems = new MenuItem[6];

        for (int i = 0; i < 6; i++)
        {
            menuItems[i] = new MenuItem();
            menuItems[i].Generate();
        }

        price1.Text = menuItems[0].Price;
        item1.Text = menuItems[0].Description;
        price2.Text = menuItems[1].Price;
        item2.Text = menuItems[1].Description;
        price3.Text = menuItems[2].Price;
        item3.Text = menuItems[2].Description;
        price4.Text = menuItems[3].Price;
        item4.Text = menuItems[3].Description;
        price5.Text = menuItems[4].Price;
        item5.Text = menuItems[4].Description;
        price6.Text = menuItems[5].Price;
        item6.Text = menuItems[5].Description;

        MenuItem guacamoleMenuItem = new MenuItem();
        guacamoleMenuItem.Generate();
        guacamole.Text = "Add guacamole for " + guacamoleMenuItem.Price;
    }
}
```

Call the MakeTheMenu method as soon as the page loads.

This array will hold 6 references to MenuItem objects.

You'll use a for loop to create each MenuItem and call its Generate method. You'll write that Generate method in the next execrise.

Now that you have the MenuItem objects with sandwich descriptions and prices, you can use them to set the text in the Label controls.

Create one more MenuItem object to generate a random price for the guacamole. You won't use its Description field.

Exercise

MenuItem
Proteins Condiments Breads Description Price
Generate

Create the MenuItem class for your menu app.

Start bhy looking closely at the class diagram. It has five fields: three arrays to hold the various sandwich parts, a description, and a price. The array fields use collection initializers, which let you define the items in an array by putting them inside curly braces.

Add the MenuItem class to your project. Here's the code for the fields:

```
class MenuItem
{
    public string[] Proteins = {
                "Roast beef", "Salami", "Turkey",
                "Ham", "Pastrami", "Tofu"
    };

    public string[] Condiments = {
                "yellow mustard", "brown mustard",
                "honey mustard", "mayo", "relish", "French dressing"
    };

    public string[] Breads = { "rye", "white", "wheat", "pumpernickel", "a roll" };

    public string Description = "";
    public string Price = "";

    public void Generate()
    {
        // You'll fill in this method
    }
}
```

The Generate method uses Random.Shared to choose random prices between 5.00 and 14.99 by creating a random decimal value out of two ints. We gave you the last line of code for the method:

```
Price = price.ToString("c");
```

The parameter to the ToString method is a format. In this case, the "c" format tells ToString to format the value with the local currency: if you're in the United States you'll see a $; in the UK you'll get a £, in the EU you'll see €, etc. If the values don't make sense in your currency, choose different random numbers!

Your job is to fill in the Generate method. It does the following:

- Picks a random protein from the Proteins array
- Picks a random condiment from the Condiments array
- Picks a random bread from the Breads array
- Sets the description field like this: `protein + " with " + condiment + " on " + bread`
- Sets the Price field to a random price that's at least 5.00 and less than 15.00. Pick a random int that's at least 5 and less than 15. Then pick a second random in that's at least 0 and less than 100. Multiply the second number by .01M to get a decimal value that's at least .00 and less than 1.00, and add it to the first value, and store it in a variable called `price`. Then set the Price field like this: `Price = price.ToString("c");`

Sharpen your pencil

Can you write a single line of code that sets Price to a random value between 5.00 and 14.99? Here's a hint: if the NextDouble method returns a value between 0 and 1. Try multiplying it by 10. What do you get?

....

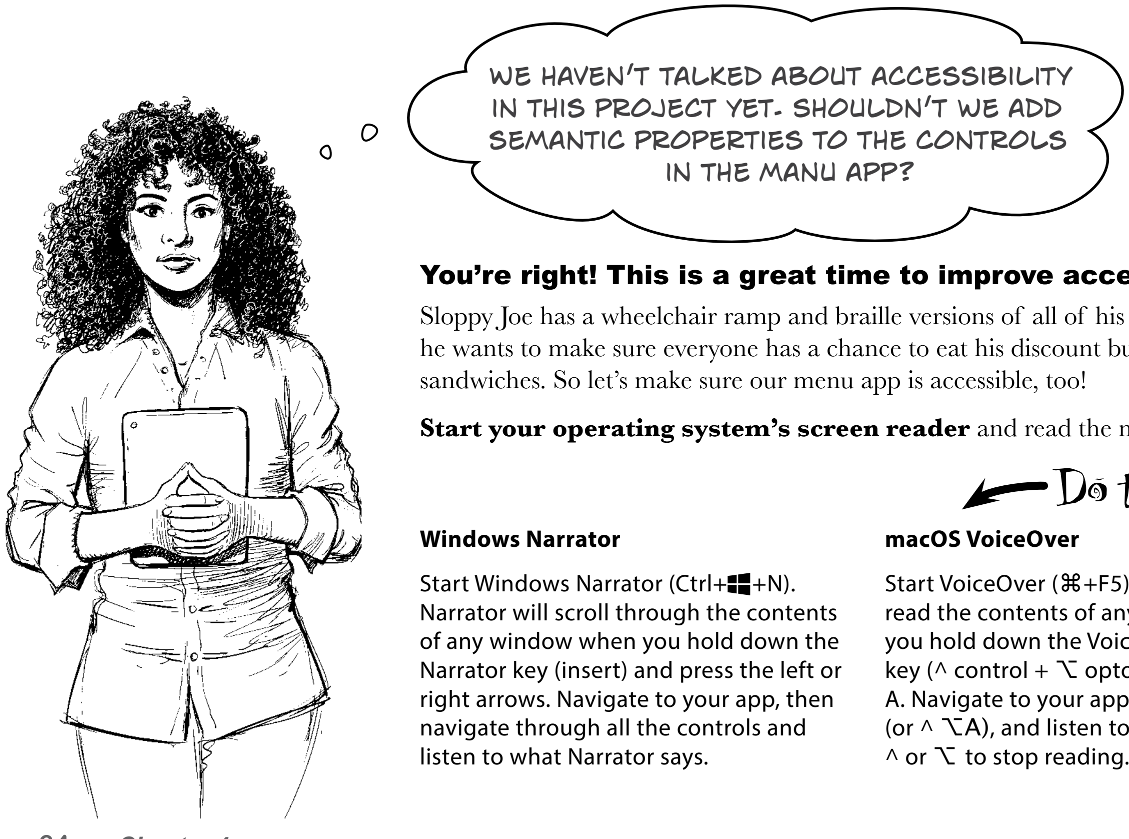
Exercise Solution

```
public void Generate()
{
    string protein = Proteins[Random.Shared.Next(Proteins.Length)];
    string condiment = Condiments[Random.Shared.Next(Condiments.Length)];
    string bread = Breads[Random.Shared.Next(Breads.Length)];
    Description = protein + " with " + condiment + " on " + bread;

    int bucks = Random.Shared.Next(5, 15);
    int cents = Random.Shared.Next(0, 100);
    decimal price = bucks + (cents * .01M);
    Price = price.ToString("c");
}
```

Can you write a single line of code that sets Price to a random value between 5.00 and 14.99? Here's a hint: if the NextDouble method returns a value between 0 and 1. Try multiplying it by 10. What do you get?

```
Price = (Random.Shared.NextDouble() * 10 + 5).ToString("c");
```


WE HAVEN'T TALKED ABOUT ACCESSIBILITY IN THIS PROJECT YET. SHOULDN'T WE ADD SEMANTIC PROPERTIES TO THE CONTROLS IN THE MANU APP?

You're right! This is a great time to improve accesibility.

Sloppy Joe has a wheelchair ramp and braille versions of all of his menus, because he wants to make sure everyone has a chance to eat his discount budget-friendly sandwiches. So let's make sure our menu app is accessible, too!

Start your operating system's screen reader and read the menu page.

Do this!

Windows Narrator

Start Windows Narrator (Ctrl+⊞+N). Narrator will scroll through the contents of any window when you hold down the Narrator key (insert) and press the left or right arrows. Navigate to your app, then navigate through all the controls and listen to what Narrator says.

macOS VoiceOver

Start VoiceOver (⌘+F5). VoiceOver will read the contents of any window when you hold down the VoiceOver activation key (^ control + ⌥ optoin) and pressing A. Navigate to your app and press VO+A (or ^ ⌥A), and listen to. Press the either ^ or ⌥ to stop reading.

Can we make the app more accessible?

When a screen reader narrates a window, it navigates from item to item, reading each item aloud and drawing a rectangle around it. What did you hear when you listened to the screen reader narrate your app? What did you see? Try having it read the menu while you have your eyes closed. Did you still understand everything that you needed to? It's pretty good! But accessibility is all about making things better for all of our users. Can we make it better?

Set the main header so the screen reader narrates it

You may have notied that the first thing it said was "Home" – and if you watched carefully, you saw that was narrating the title bar. **Modify AppShell.xaml to change "Home" to "Sloppy Joe's menu"** and have the screen reader narrate the page again. When you're

It would be great to have the narrator tell the user that they're looking at items on a menu. Let's try adding a SemanticProperties.Description to the <Grid> tag:

```
<Grid Margin="10"
      SemanticProperties.Description="Here are the items on the menu.">
```

Now try using the screen reader to narrate the window. It sounds fine in Windows, but if you're using macOS there's a problem: the screen reader won't read the items or prices. That's because if you set the SemanticProperties.Description on a that has children, the screen reader can't reach those children anymore. This is important even if you're building software for Windows, because your MAUI apps are cross-platform, and you want your app to be accessible anywhere.

Try setting the item1 label's SemanticProperties.Description instead

Ok, let's try something else. Remove the SemanticProperties.Description property from the <Grid> tag. Then try setting the SemanticProperties.Description on the first label:

```
   <Label x:Name="item1" FontSize="18" Text="item #1"
          SemanticProperties.Description="Here are the items on the menu.">
```

Try using the screen reader again. ***It's still not right!*** When you have a Label, you always want the screen reader to read the contents of the label. Setting the SemanticProperties.Description causes the screen reader to read that description <u>instead of the label text</u>.

Go ahead and delete the SemanticDescription property from the item1 Label control (and also from the Grid, if you haven't done it already).

Brain Power

What do you think you'll do to make the screen reader say "Here are the items on the menu" followed by the randomly generated sandwich in the item1 Label control's Text property?

Use the SetValue method to change a control's semantic properties

Let's find a different way to make the screen reader say "Here are the items on the menu." before it reads the menu items. We'll still use the SemanticProperties.Description for the first menu item, but instead of using a XAML tag, we'll use C# to make sure it preserves the text.

Add this line of code to the end of your MainPage method:

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

    item1.SetValue(SemanticProperties.DescriptionProperty,
        "Here are the items on the menu. " + item1.Text);
}
```

If you type "item1." into Visual Studio, you won't see SemanticProperties in the IntelliSense pop-up. That's why you need to use the SetValue method to set it instead.

This code sets the SemanticProperties.Description property—in this case, it's setting it to the text "Here are the items on the menu" followed by the random sandwich generated by MenuItem. Try the screen reader one more time—now the page includes that text, and works on all operating systems.

Bullet Points

- The **new keyword** returns a reference to an object that you can store in a reference variable.
- You can have **multiple references** to the same object. You can change an object with one reference and access the results of that change with another.
- For an object to stay in the heap, it **has to be referenced**. Once the last reference to an object disappears, it eventually gets **garbage-collected** and the memory it used is reclaimed.
- Your .NET apps run in the **Common Language Runtime** (CLR), a "layer" between the OS and your program. The C# compiler builds your code into **Common Intermediate Language** (CIL), which the CLR executes.
- Declare **array variables** by putting square brackets after the type in the variable declaration (like bool[] trueFalseValues or Dog[] kennel).
- Use the **new keyword to create a new array**, specifying the array length in square brackets (like new bool[15] or new Dog[3]).The **this keyword** lets an object get a reference to itself.
- Use the **Length method** on an array to get its length (like kennel.Length).
- Access an array value using its **index** in square brackets (like bool[3] or Dog[0]). Array indexes **start at 0**.
- null means a reference **points to nothing**. The compiler will warn you when a variable can **potentially be null**.
- Use the **string? type** to hold a string that's allowed be null. Console.ReadLine can return null strings.
- Use **collection initializers** to initialize an array by setting the array equal to the new keyword followed by the array type followed by a comma-delimited list in curly braces (like new int[] { 8, 6, 7, 5, 3, 0, 9 }). The array type is optional when setting a variable or field value in the same statement where it's declared.
- You can pass a **format parameter** to an object or value's ToString method. If you're calling a numeric type's ToString method, passing it a value of "c" formats the value as a local currency.
- Use a control's **SetValue method** to set its semantic properties in code, so the screen reader can include text that's generated when the app runs.