Customize Layouts

Introduction

- .NET MAUI layout panels help you create consistent user interfaces for your application across a wide range of devices.

- Designing a user interface that's consistent across multiple devices is difficult because devices can be different sizes and have different pixel densities.
- Think about the different devices that are available: mobile, tablet, desktop, and so on.
- How do we create a user interface that looks similar on each?

- .NET MAUI provides layout panels to help you build consistent user interfaces.
- The layout panel is responsible for sizing and positioning the views of its children.

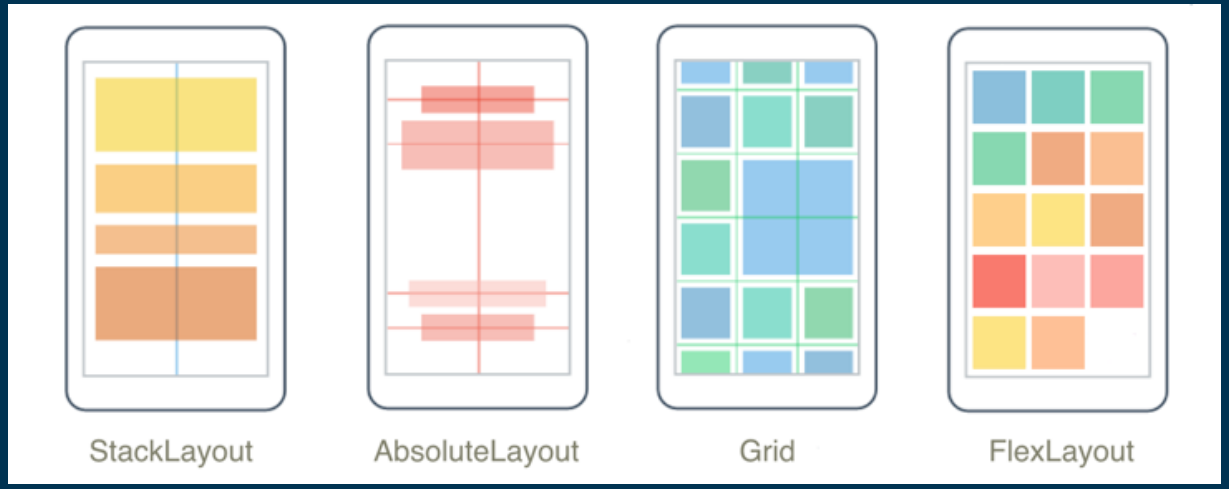
What is a Layout Panel?

- A layout panel is a .NET MAUI container that holds a collection of child views and determines their size and position.
- The layout panels automatically recalculate when the app size changes; for example, when the user rotates the device.

- **Note:**
  - The term view or child view refers to a control placed on a layout panel.
  - A view can be a label, a button, an entry field, or any other type of visual element supported by .NET MAUI.

What is a Layout Panel?

- .NET MAUI has multiple layout panels that you can choose from.
- Each panel manages its child views differently.

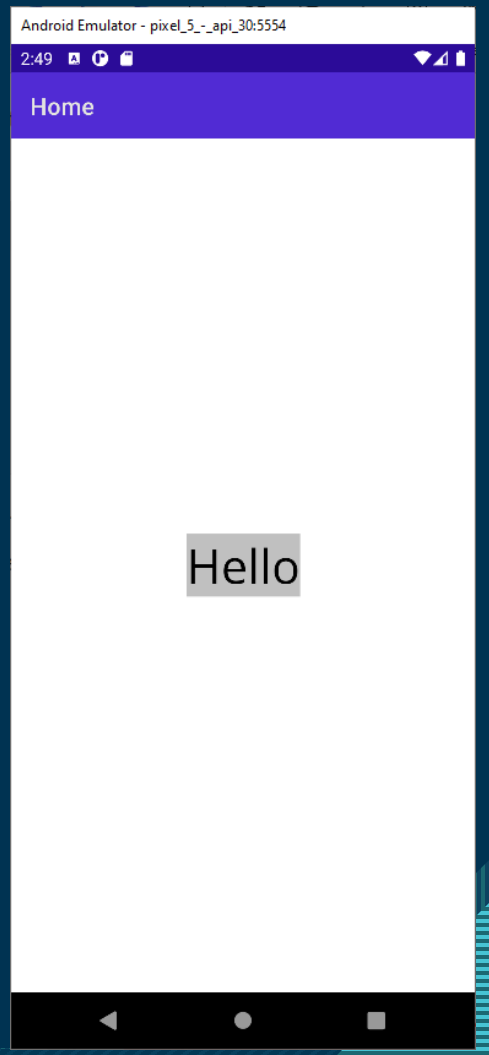


- `StackLayout`: arranges its child views in a single row or column.
  - In addition to `StackLayout`, there's also a new optimized `VerticalStackLayout` and `HorizontalStackLayout` when you don't need to change orientation.
- `AbsoluteLayout`: arranges its child views by using x and y coordinates.
- `Grid`: arranges its child views in cells that are created from the intersection of rows and columns.
- `FlexLayout`: arranges its child views like a `StackLayout` except that you can wrap them if they don't fit into a single row or column.

Default Size of a View

- If you don't specify the size of a view, it grows automatically to be exactly large enough to fit around its content.
- For example, consider this XAML:

```
<Label
    Text="Hello"
    BackgroundColor="Silver"
    VerticalOptions="Center"
    HorizontalOptions="Center"
    FontSize="40"/>
```

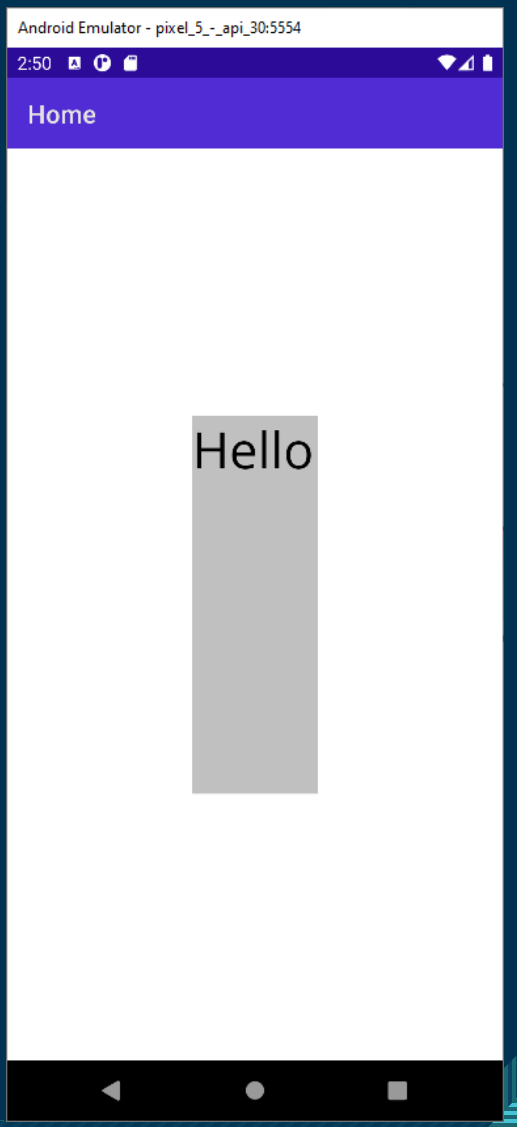


- It defines a label to display the word Hello on a silver background.
- Because you're not specifying the size, the label is automatically sized to fit around the word Hello.

Specify the Size of a View

- The `View` base class defines two properties that influence the size of a view:
  - `WidthRequest`: lets you specify the width
  - `HeightRequest`: lets you specify the height

```
<Label
    Text="Hello"
    BackgroundColor="Silver"
    VerticalOptions="Center"
    HorizontalOptions="Center"
    WidthRequest="100"
    HeightRequest="300"
    FontSize="40"/>
```



Specify the Size of a View

- One thing that's worth noting is the names of these properties.
- Both properties contain the word request.
- This word means the layout panel might not respect them at runtime.
- The layout panel reads these values during its sizing calculations and tries to accommodate the requests if it can.
- If there's not enough space, the layout panel is allowed to ignore the values.

Size Units

- When you set `WidthRequest` and `HeightRequest`, you use literal values like `100`.
- At the .NET MAUI level, these values don't have units.
  - They're not points or pixels.
  - They're just values of type double.

- .NET MAUI passes these values to the underlying operating system at runtime.
- It's the operating system that provides the context needed to determine what the numbers mean.

- On iOS, the values are called points.
- On Android, they're density-independent pixels.

Specify the Position of a View

- The `View` base class has two properties that you use to set the position of a view:
  - `VerticalOptions`
  - `HorizontalOptions`

- These settings influence how the view is positioned within the layout panel.
- You can specify that you want the view to align to one of the four edges of the layout panel.
- Or, that you want it to occupy the entire layout panel.

Exercise: Explore Alignment Options

- In this exercise, you use a .NET MAUI application to see the effect of the four primary alignment options when applied to a view contained in a `Grid`.
- You don't write code in the exercise.
- Instead, you use the provided solution and select buttons to change the layout options of a label.

- This module uses the .NET 8.0 SDK.
- Ensure that you have it installed.

- Clone or download the exercise repo from GitHub:
  - https://github.com/microsoftdocs/mslearn-dotnetmaui-customize-xaml-pages-layout
  - Open the starter solution from the exercise1/Alignment folder by using Visual Studio.

Exercise: Explore Alignment Options

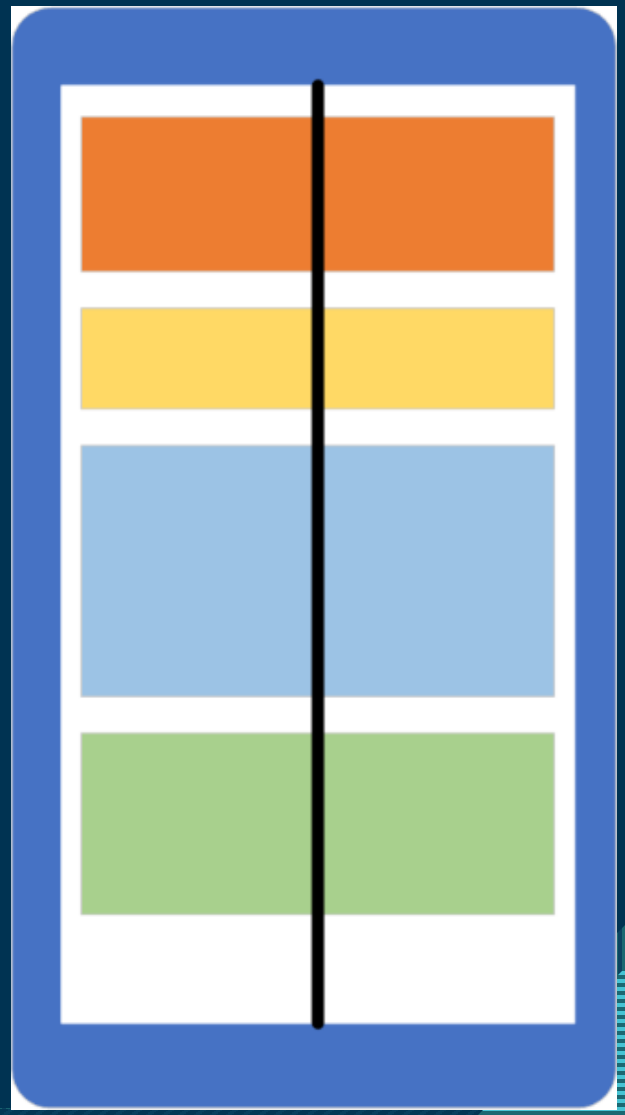
- Run the app to test `LayoutOptions` and see how the different layout options change the size and position of the label.
- Test the app by interacting with the buttons that change the horizontal and vertical `LayoutOptions`.

- This image shows what happens if you select Center for both the `Horizontal` and `Vertical` alignment options.

Arrange Views with StackLayout

- `StackLayout` is a layout container that organizes its child views left-to-right or top-to-bottom.
- The direction is based on its `Orientation` property, and the default value is top-to-bottom.

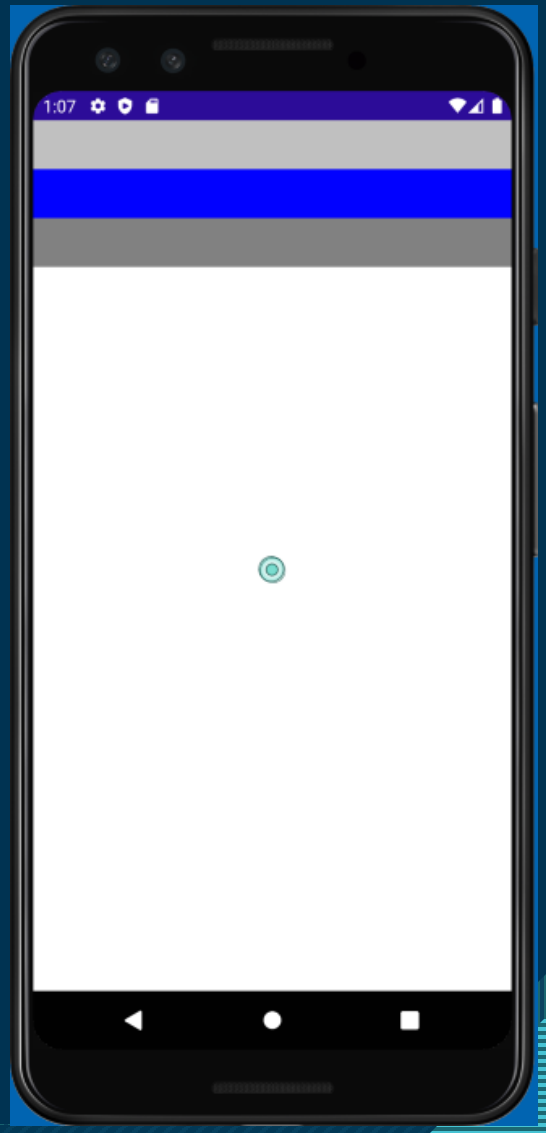
- `VerticalStackLayout` and `HorizontalStackLayout` are the preferred layouts to use when you know that your orientation isn't going to change, because they're optimized for performance.



How to Add Views to a StackLayout

- You can add child views inside the `StackLayout` tag.
- It automatically positions the views in a vertical list.

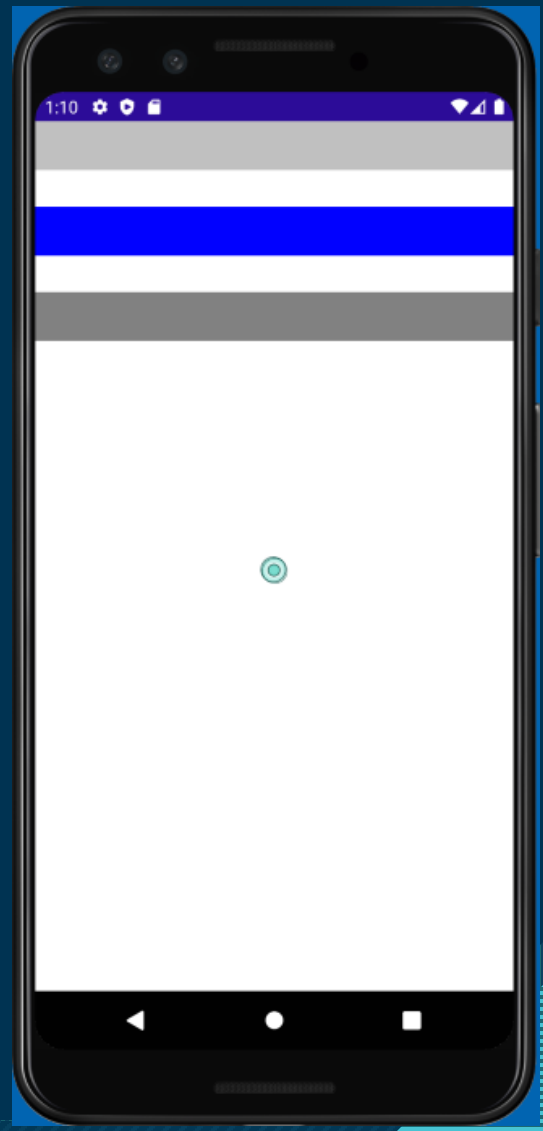
```
<StackLayout>
    <BoxView Color="Silver" />
    <BoxView Color="Blue" />
    <BoxView Color="Gray" />
</StackLayout>
```



How to Change the Space Between Views in a StackLayout

- It's common to want some space between the children of a `StackLayout`.
- `StackLayout` lets you control the space between each child by using the `Spacing` property.
- The default value is zero units, but you can set it to whatever looks good to you.

```
<StackLayout Spacing="30">
    <BoxView Color="Silver" />
    <BoxView Color="Blue" />
    <BoxView Color="Gray" />
</StackLayout>
```

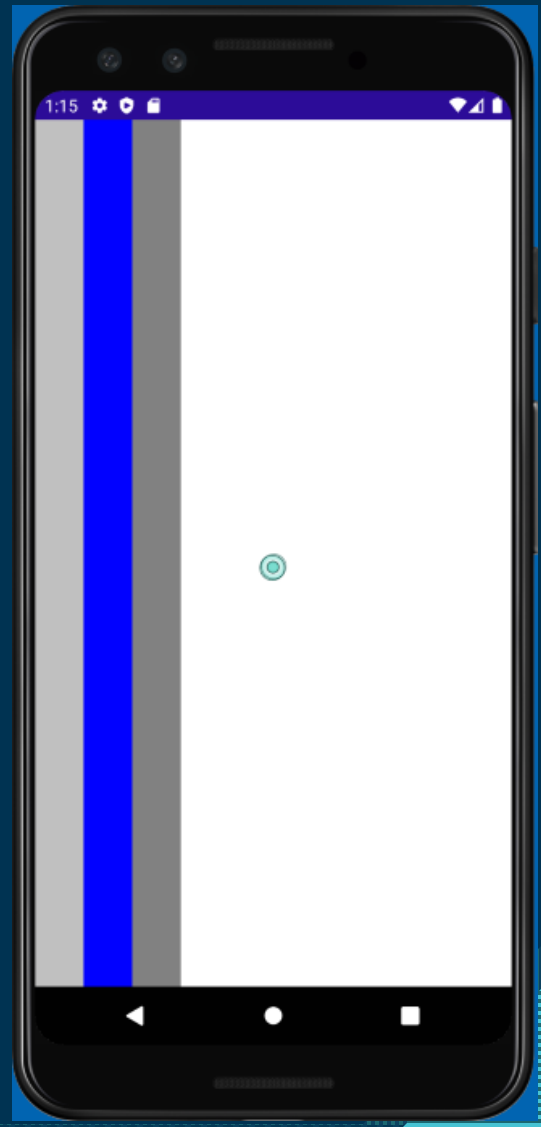


How to Set the Orientation of a StackLayout

- `StackLayout` lets you arrange children in either a column or a row.
- You control this behavior by setting its `Orientation` property.
- So far, we've been showing only a vertical `StackLayout`.

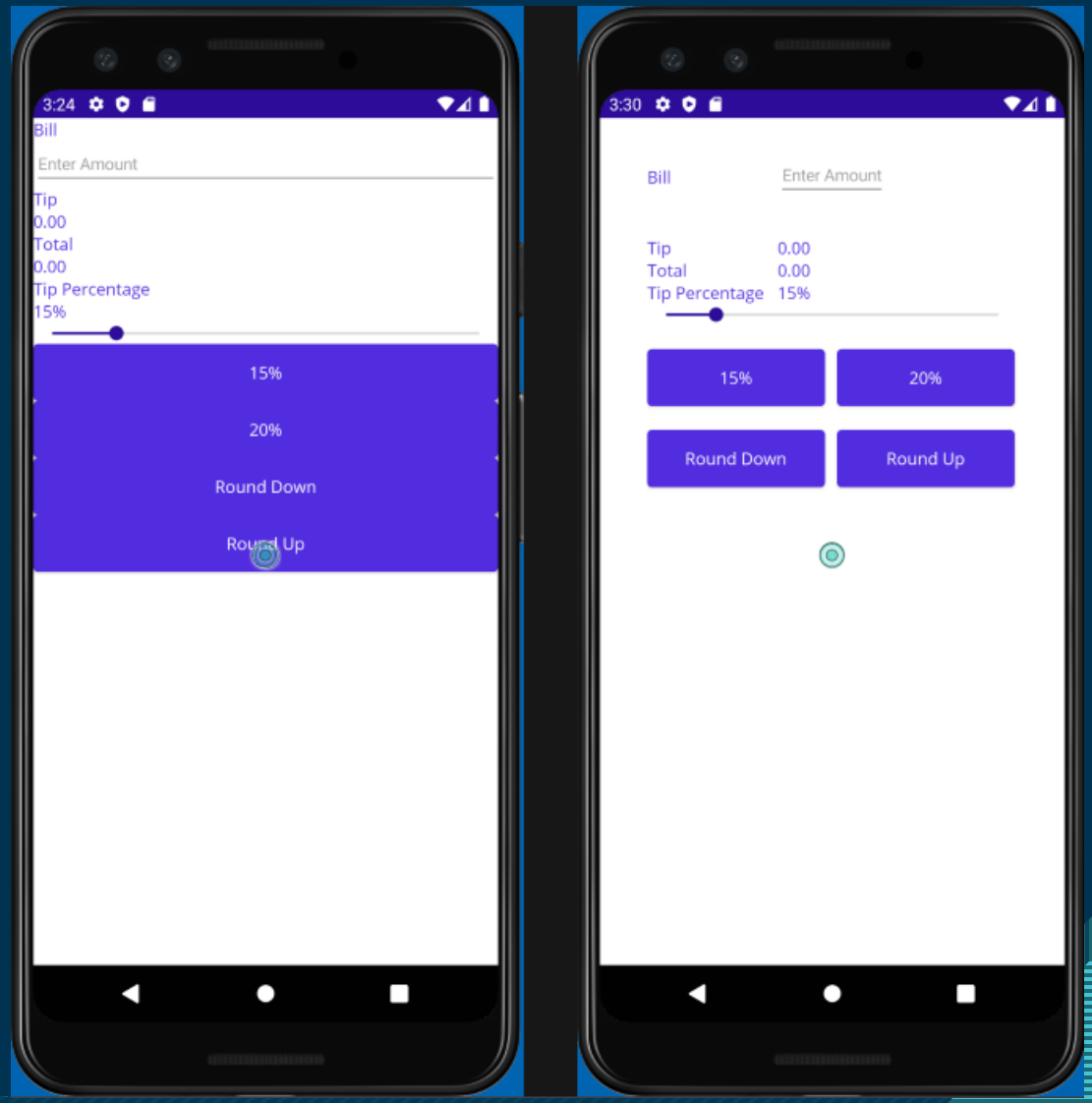
- `Vertical` is the default.

```
<StackLayout Orientation="Horizontal">
    <BoxView Color="Silver" WidthRequest="40"/>
    <BoxView Color="Blue" WidthRequest="40"/>
    <BoxView Color="Gray" WidthRequest="40"/>
</StackLayout>
```



Exercise: Use StackLayout to Build a User Interface

- In this exercise, you use nested `StackLayout` containers to arrange the views in your User Interface (UI).
- The first screenshot shows the layout implemented by the starter project, and the second one shows the layout for the completed project.
- Your job is to use `StackLayout` containers and `LayoutOptions` to turn the starter project into the completed version.



Exercise: Explore the Starter Solution

- The starter solution contains a fully functional tip calculator app.
- Start by exploring the UI to understand what the app does.
- Using Visual Studio, open the starter solution in the exercise2/TipCalculator folder in the repo that you cloned at the start of the previous exercise.
- Build and run the app on your preferred operating system.
- Enter a number into the text box and use the app to see how it works.
- Experiment with the tip amount buttons and the slider.
- When you're finished, close the app.

Exercise: Explore the Starter Solution

- Open MainPage.xaml. Notice that all the views are placed into one `VerticalStackLayout`.

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:TipCalculator"
             x:Class="TipCalculator.MainPage">
    <VerticalStackLayout>
        <Label Text="Bill" />
        <Entry x:Name="billInput" Placeholder="Enter Amount" Keyboard="Numeric" />

        <Label Text="Tip"   />
        <Label x:Name="tipOutput" Text="0.00" />

        <Label Text="Total" />
        <Label x:Name="totalOutput" Text="0.00" />

        <Label Text="Tip Percentage" />
        <Label x:Name="tipPercent" Text="15%" />
        <Slider x:Name="tipPercentSlider" Minimum="0" Maximum="100" Value="15" />

        <Button Text="15%" Clicked="OnNormalTip" />
        <Button Text="20%" Clicked="OnGenerousTip" />

        <Button x:Name="roundDown" Text="Round Down" />
        <Button x:Name="roundUp"   Text="Round Up" />
    </VerticalStackLayout>
</ContentPage>
```

Exercise: Fix the UI

- Now that you saw the app run, you can make it look better by adding `HorizontalStackLayout` containers.
- The goal is to make the app look like the screenshot at the start of the lab.
- Open the MainPage.xaml file.
- Add 40 units of padding and 10 units of spacing to the `VerticalStackLayout`.

```
<VerticalStackLayout Padding="40" Spacing="10">
```

Exercise: Fix the UI

- Add a `HorizontalStackLayout` to group the `Label` that says Bill with the `Entry` field below it.
- Set the `Spacing` property to 10.
- Set the `WidthRequest` of the Bill `Label` to 100 and the `VerticalOptions` property to Center.
- These changes ensure that the label is aligned vertically with the `Entry` field.

```
<HorizontalStackLayout Spacing="10">
    <Label Text="Bill" WidthRequest="100" VerticalOptions="Center"/>
    <Entry ... />
</HorizontalStackLayout>
```

Exercise: Fix the UI

- Add another `HorizontalStackLayout` to group the `Label` that says Tip with the `Label` named tipOutput.
- Set the `Spacing` property to `10`, and the `Margin` property to `0,20,0,0`.
- Set the `WidthRequest` of the Tip `Label` to `100`.

```
<HorizontalStackLayout Margin="0,20,0,0" Spacing="10">
    <Label Text="Tip" WidthRequest="100" />
    <Label .../>
</HorizontalStackLayout>
```

Exercise: Fix the UI

- Use a `HorizontalStackLayout` to group the `Label` that says Total with the `Label` named totalOutput.
- Set the `Spacing` property to 10.
- Set the `WidthRequest` of the Total `Label` to 100.

```
<HorizontalStackLayout Spacing="10">
    <Label Text="Total" WidthRequest="100"  />
    <Label .../>
</HorizontalStackLayout>
```

Exercise: Fix the UI

- Add another `HorizontalStackLayout` to group the `Label` that says Tip Percentage with the `Label` named tipPercent.
- Set the `VerticalOptions` property of this `HorizontalStackLayout` to `End` and set the `Spacing` property to `10`.
- Set the `WidthRequest` of the Tip Percentage `Label` to `100`.

```
<HorizontalStackLayout VerticalOptions="End" Spacing="10">
    <Label Text="Tip Percentage" WidthRequest="100"/>
    <Label ... />
</HorizontalStackLayout>
```

Exercise: Fix the UI

- Use a `HorizontalStackLayout` to group the `Button` with the caption `15%` and the `Button` with the caption `20%`.
- Set the `Margin` property of this `StackLayout` to `0,20,0,0`, and the `Spacing` property to `10`.

```
<HorizontalStackLayout  Margin="0,20,0,0" Spacing="10">
    <Button Text="15%" ... />
    <Button Text="20%" ... />
</HorizontalStackLayout>
```

Exercise: Fix the UI

- Add a final `HorizontalStackLayout` to group the `Button` with the caption, Round Down and the `Button` with the caption, Round Up.
- Set the `Margin` property of this `StackLayout` to `0,20,0,0`, and the `Spacing` property to `10`.

```
<HorizontalStackLayout Margin="0,20,0,0" Spacing="10">
    <Button ... Text="Round Down" />
    <Button ... Text="Round Up" />
</HorizontalStackLayout>
```

Exercise: Fix the UI

- On all four button controls, set the `HorizontalOptions` property to `Center` and the `WidthRequest` property to `150`.
- For example:

```
<Button Text="15%" WidthRequest="150" HorizontalOptions="Center" ... />
```

Exercise: Fix the UI

- The complete XAML markup for the content page should look like 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"
             xmlns:local="clr-namespace:TipCalculator"
             x:Class="TipCalculator.MainPage">
    <VerticalStackLayout Padding="40" Spacing="10">
        <HorizontalStackLayout Spacing="10">
            <Label Text="Bill" WidthRequest="100" VerticalOptions="Center" />
            <Entry x:Name="billInput" Placeholder="Enter Amount" Keyboard="Numeric" />
        </HorizontalStackLayout>
        <HorizontalStackLayout Margin="0,20,0,0" Spacing="10">
            <Label Text="Tip"  WidthRequest="100" />
            <Label x:Name="tipOutput" Text="0.00" />
        </HorizontalStackLayout>
        <HorizontalStackLayout Spacing="10">
            <Label Text="Total" WidthRequest="100"/>
            <Label x:Name="totalOutput" Text="0.00" />
        </HorizontalStackLayout>
        <HorizontalStackLayout VerticalOptions="End" Spacing="10">
            <Label Text="Tip Percentage" WidthRequest="100"/>
            <Label x:Name="tipPercent" Text="15%" />
        </HorizontalStackLayout>
        <Slider x:Name="tipPercentSlider" Minimum="0" Maximum="100" Value="15" />
        <HorizontalStackLayout Margin="0,20,0,0" Spacing="10">
            <Button Text="15%" Clicked="OnNormalTip" WidthRequest="150" HorizontalOptions="Center"/>
            <Button Text="20%" Clicked="OnGenerousTip"  WidthRequest="150" HorizontalOptions="Center"/>
        </HorizontalStackLayout>
        <HorizontalStackLayout Margin="0,20,0,0" Spacing="10">
            <Button x:Name="roundDown" Text="Round Down" WidthRequest="150" HorizontalOptions="Center"/>
            <Button x:Name="roundUp"   Text="Round Up" WidthRequest="150" HorizontalOptions="Center"/>
        </HorizontalStackLayout>
    </VerticalStackLayout>
</ContentPage>
```

Exercise: Examine the Results

- Run the app again and look at the differences in the UI.
- Verify that the controls are aligned correctly and are properly sized and spaced.

Arrange Views with Grid

- A `Grid` is a layout panel that consists of rows and columns.
- You place views in the cells that are created from the intersection of the rows and columns.
- For example, if you create a `Grid` that has three columns and two rows, there are six cells available for views.

How to Specify the Rows and Columns of a Grid

- When you create a `Grid`, you can define each row and column individually.
- Every `Grid` has a collection of `RowDefinition` and `ColumnDefinition` objects that define the shape of the grid.
- You populate these collections with instances of `RowDefinition` and `ColumnDefinition`, each representing a row or column in your UI.

- `RowDefinition` has a property called `Height` and `ColumnDefinition` has a property called `Width`.

What is GridLength?

- The data type for the `Width` and `Height` properties is `GridLength`.
- You can set it to one of these values:
  - Absolute value
  - `Auto`
  - Star (`*`)

Absolute

- Absolute specifies that the row or column should be fixed in size.

```
<RowDefinition Height="100" />
```

Auto

- `Auto` automatically sizes the row or column to fit your child views.
- The `Grid` scans all child views in that row or column, selects the largest view, and then makes the row or column large enough to fit that child.

```
<RowDefinition Height="Auto" />
```

Star

- Star (*) gives you proportional sizing.
- In proportional sizing, the total available space and the ratio that each row or column asks for determines the size.
- In conversation, people often call this star sizing instead of proportional sizing.
- Use the * symbol to represent star sizing.

```
<RowDefinition Height="2*" />
```

Grid Collections

- After you define the rows and columns by using `RowDefinition` and `ColumnDefinition`, you can add them to a `Grid`.
- You use the `RowDefinitions` and `ColumnDefinitions` collection properties of the `Grid`.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="100" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="1*" />
        <RowDefinition Height="2*" />
    </Grid.RowDefinitions>
    ...
</Grid>
```

- This definition can be shortened to:

```
<Grid RowDefinitions="100, Auto, 1*, 2*">
    ...
</Grid>
```

Row and Column Default Size

- The default for rows and columns is 1* size.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    ...
</Grid>
```

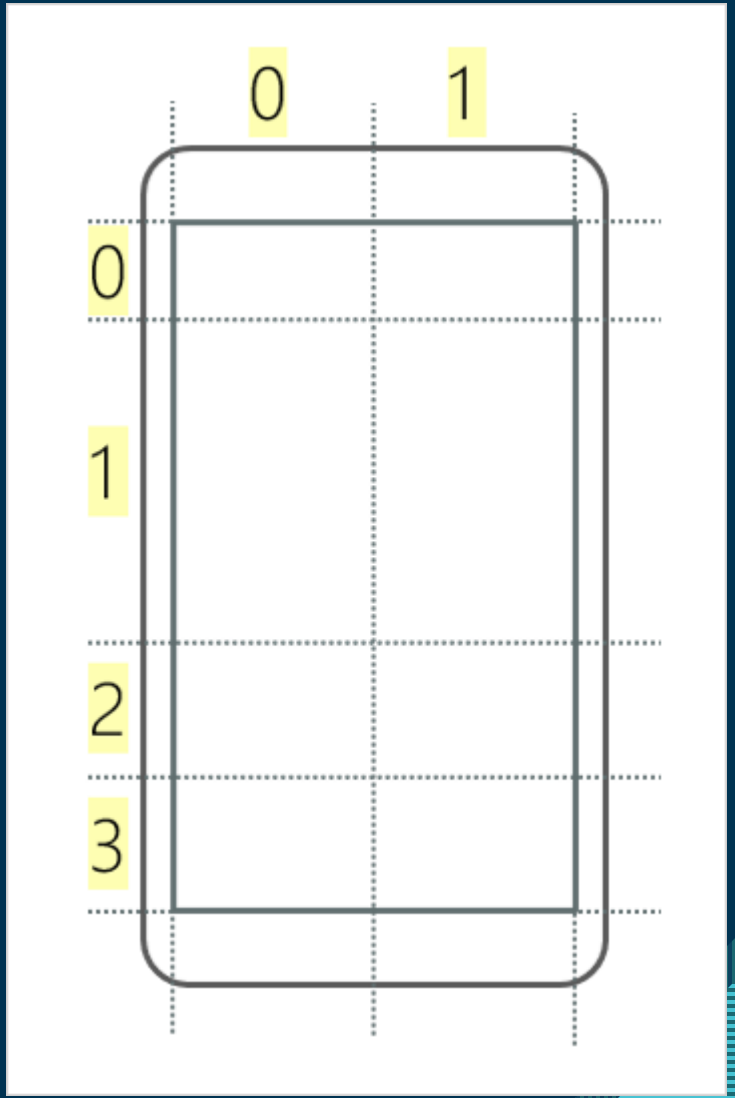
- This definition can be shortened to:

```
<Grid RowDefinitions="*, *, *" ColumnDefinitions="*, *">
    ...
</Grid>
```

How to Add Views to a Grid

- When you add a view to a `Grid`, you add it to a specific cell.
- You use a combination of a row number and a column number to identify a cell.
- The numbering of rows and columns starts at zero.

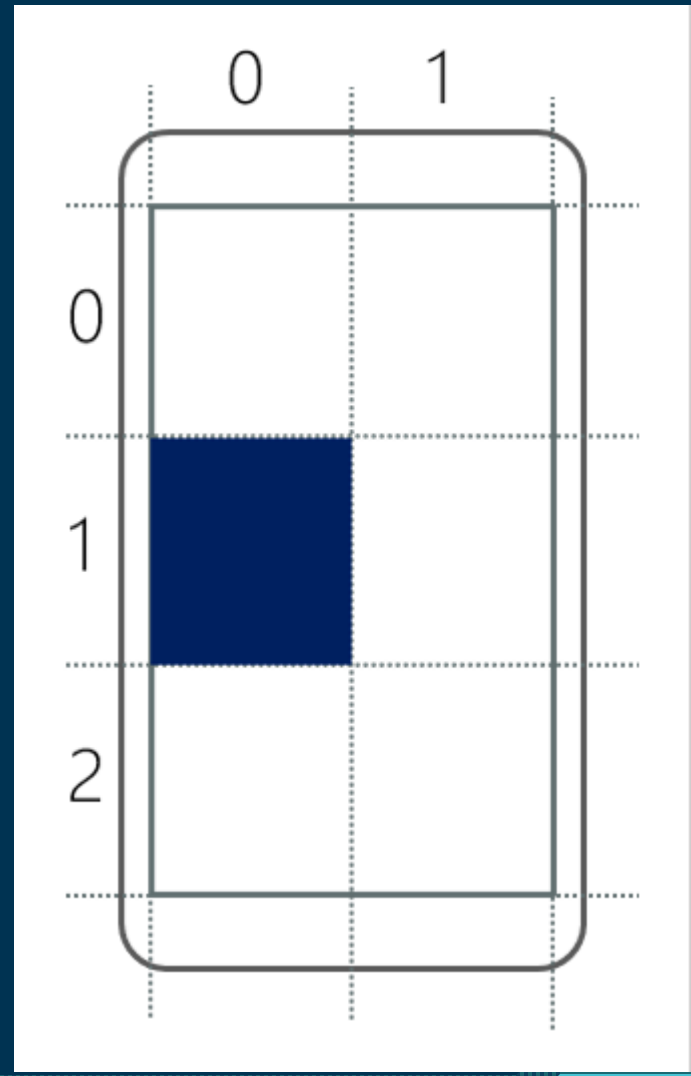
- For example, if we wanted to add a view to the bottom-right cell, we'd say the view's position was row 3 column 1.



Add a View to a Grid by using Attached Properties

- Attached properties are a collection of key-value pairs that is part of a view.
- When you add a view to a `Grid`, you specify the row and column.
- By using attached properties, you can add a key-value pair with the key `Grid.Row` and a value that specifies the row number.

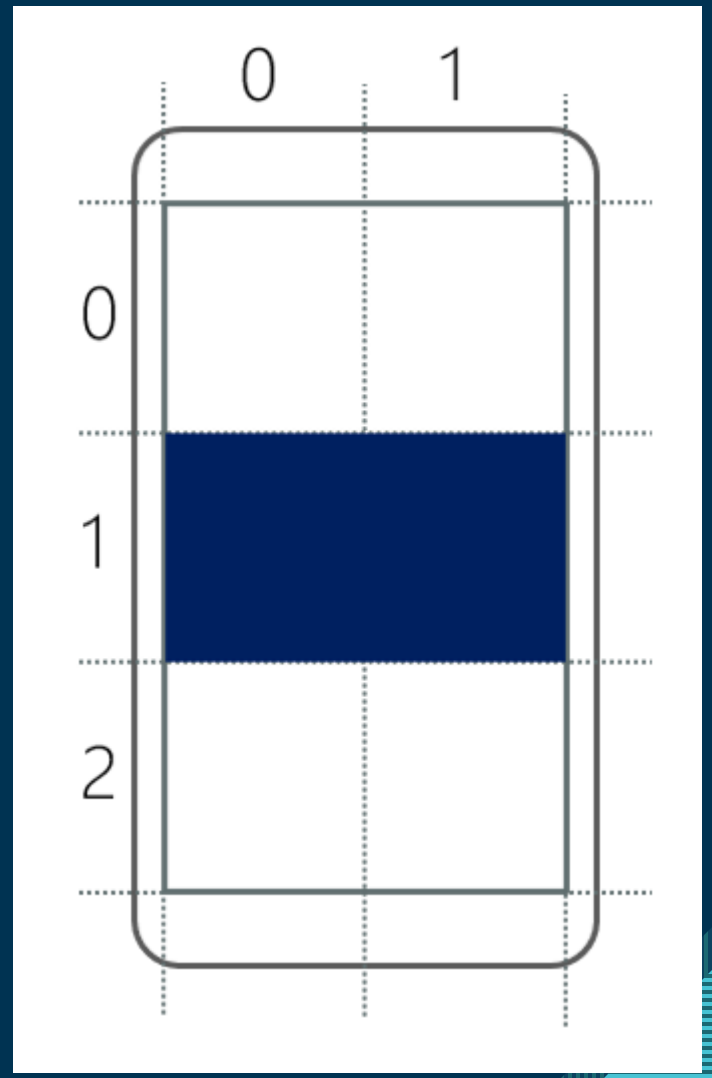
```
<Grid RowDefinitions="*, *, *" ColumnDefinitions="*, *">

    <BoxView Grid.Row="1" Grid.Column="0" Color="Navy" />

</Grid>
```



How to Make a View Span Multiple Rows or Columns

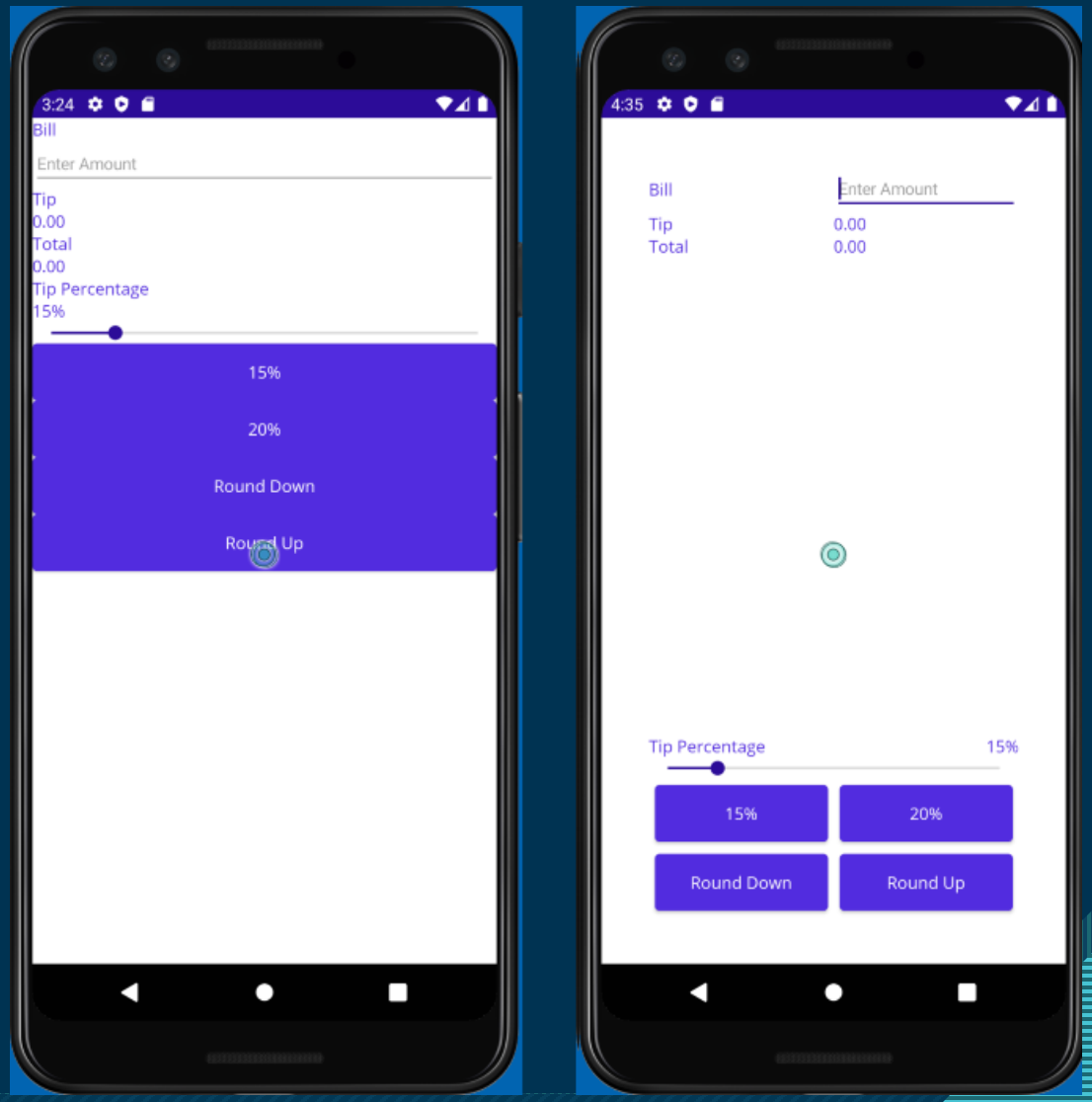
- There are two more attached properties you should be aware of: `Grid.RowSpan` and `Grid.ColumnSpan`.
- These properties specify how many rows or columns the view should occupy.

```
<Grid RowDefinitions="*, *, *" ColumnDefinitions="*, *">

    <BoxView Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" Color="Navy" />

</Grid>
```



Exercise: Use Grid to Build a User Interface

- In this exercise, you use a `Grid` to arrange the views in your User Interface (UI).
- You start with another version of the TipCalculator project and adjust it to make the UI more intuitive.
- You also move the buttons to the bottom of the page.
- This time you use a `Grid` layout rather than using `VerticalStackLayout` and `HorizontalStackLayout`.



Exercise: Open the Starter Solution

- The starter solution contains a fully functional tip calculator app.
- Using Visual Studio, open the starter solution in the exercise3/TipCalculator folder in the repo that you cloned at the start of the previous exercise.

Exercise: Open the Starter Solution

- Open MainPage.xaml. Notice that all the views are displayed using one `VerticalStackLayout` panel.

```
<?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"
             xmlns:local="clr-namespace:TipCalculator"
             x:Class="TipCalculator.MainPage">

    <VerticalStackLayout>

        <Label Text="Bill" />
        <Entry x:Name="billInput" Placeholder="Enter Amount" Keyboard="Numeric" />

        <Label Text="Tip"   />
        <Label x:Name="tipOutput" Text="0.00" />

        <Label Text="Total" />
        <Label x:Name="totalOutput" Text="0.00" />

        <Label Text="Tip Percentage" />
        <Label x:Name="tipPercent" Text="15%" />
        <Slider x:Name="tipPercentSlider" Minimum="0" Maximum="100" Value="15" />

        <Button Text="15%" Clicked="OnNormalTip" />
        <Button Text="20%" Clicked="OnGenerousTip" />

        <Button x:Name="roundDown" Text="Round Down" />
        <Button x:Name="roundUp"   Text="Round Up" />

    </VerticalStackLayout>

</ContentPage>
```

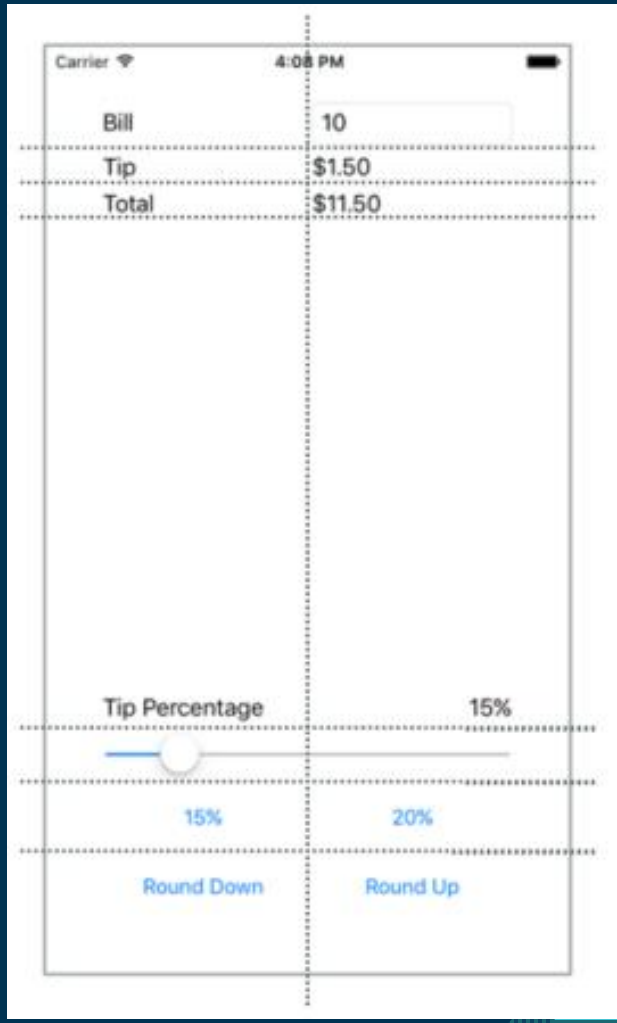
Exercise: Create a Grid Layout

- Change the layout panel from `VerticalStackLayout` to `Grid` with padding of 40 units.
- Define seven rows and two columns for the `Grid`.
- Make all the rows Auto size except the fourth row.
- The fourth row should use Star so it gets all the remaining space available in the grid.
- Use Star sizing for both columns.

```
<Grid RowDefinitions="Auto, Auto, Auto, *, Auto, Auto, Auto"
      ColumnDefinitions="*, *"
      Padding="40">
    ...
</Grid>
```

Exercise: Position the Views in the Cells

- Add settings for `Grid.Row` and `Grid.Column` to each of the views to assign them to the appropriate cell in the `Grid`.
- Use this screenshot to help you determine where each view should be placed.



- The following example shows how to set the position for the Bill `Label`, and the billInput `Entry` view:

```
...
<Label Text="Bill" Grid.Row="0" Grid.Column="0"/>
<Entry x:Name="billInput" Placeholder="Enter Amount"
Keyboard="Numeric" Grid.Row="0" Grid.Column="1"/>
...
```

Exercise: Position the Views in the Cells

- Align the Bill `Label` and `Entry` by setting the `VerticalOptions` property to Center on the `Label`.

- Add a setting for `Grid.ColumnSpan` to the `Slider` so it spans two columns.

```
<Slider ... Grid.ColumnSpan="2" ... />
```

Exercise: Position the Views in the Cells

- Locate the `Label` with the text Tip Percentage.
- Set it so that it occupies the lower-left position in its rectangle.

```
<Label Text="Tip Percentage" VerticalOptions="End" HorizontalOptions="Start" ... />
```

Exercise: Position the Views in the Cells

- Locate the `Label` named tipPercent.
- Set it so that it occupies the lower-right position in its rectangle.

```
<Label x:Name="tipPercent" VerticalOptions="End" HorizontalOptions="End" ... />
```

- Set the `Margin` property for all four buttons to `5`.

Exercise: Position the Views in the Cells

- The complete XAML markup for the page should look like this:

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:TipCalculator"
             x:Class="TipCalculator.MainPage">
    <Grid RowDefinitions="Auto, Auto, Auto, *, Auto, Auto, Auto"
          ColumnDefinitions="*, *"
          Padding="40">
        <Label Text="Bill" VerticalOptions="Center" Grid.Row="0" Grid.Column="0"/>
        <Entry x:Name="billInput" Placeholder="Enter Amount" Keyboard="Numeric" Grid.Row="0" Grid.Column="1"/>
        <Label Text="Tip" Grid.Row="1" Grid.Column="0"/>
        <Label x:Name="tipOutput" Text="0.00" Grid.Row="1" Grid.Column="1"/>
        <Label Text="Total" Grid.Row="2" Grid.Column="0"/>
        <Label x:Name="totalOutput" Text="0.00" Grid.Row="2" Grid.Column="1"/>
        <Label Text="Tip Percentage" VerticalOptions="End" HorizontalOptions="Start" Grid.Row="3" Grid.Column="0"/>
        <Label x:Name="tipPercent" Text="15%" VerticalOptions="End" HorizontalOptions="End" Grid.Row="3" Grid.Column="1"/>
        <Slider x:Name="tipPercentSlider" Minimum="0" Maximum="100" Value="15" Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2"/>
        <Button Text="15%" Clicked="OnNormalTip" Margin="5" Grid.Row="5" Grid.Column="0"/>
        <Button Text="20%" Clicked="OnGenerousTip" Margin="5" Grid.Row="5" Grid.Column="1"/>
        <Button x:Name="roundDown" Margin="5" Text="Round Down" Grid.Row="6" Grid.Column="0"/>
        <Button x:Name="roundUp"   Margin="5" Text="Round Up" Grid.Row="6" Grid.Column="1"/>
    </Grid>
</ContentPage>
```

Exercise: Examine the results

- Run the application and look at the differences in the UI.
- You used a `Grid` to improve the aesthetics of an existing UI.
- `Grid` is more powerful than `StackLayout`.
- In particular, `Grid` makes it far easier to align views across rows.

Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

References

Material has been taken as is from:

- Microsoft Official Documentation:
  - https://learn.microsoft.com/en-us/training/modules/customize-xaml-pages-layout/