IOS300

Auto Layout and Constraints

Download class materials from university.xamarin.com

Microsoft

Xamarin University

Objectives

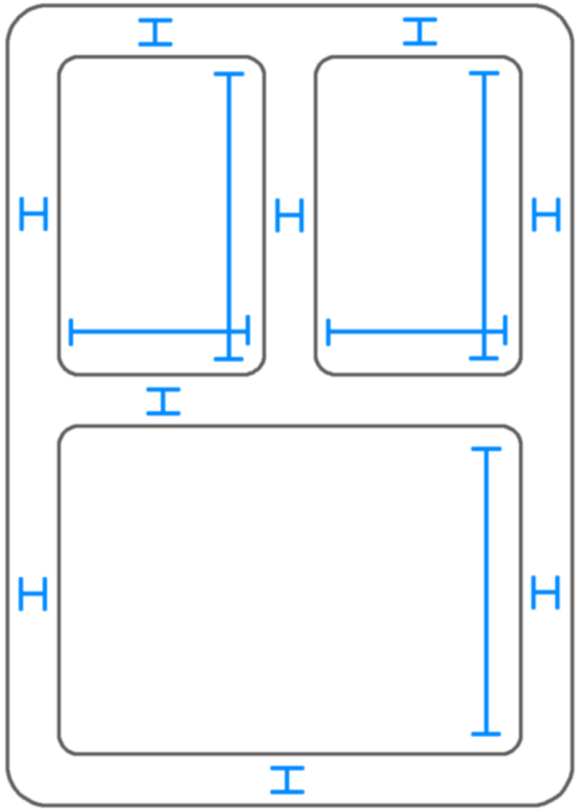
1. Create adaptive UIs
   using the iOS Designer
2. Create and update constraints
   programmatically
3. Animate constraint changes
4. Use Size Classes to customize your UI
   for different screen sizes

Create adaptive UIs using the iOS Designer

Xamarin University

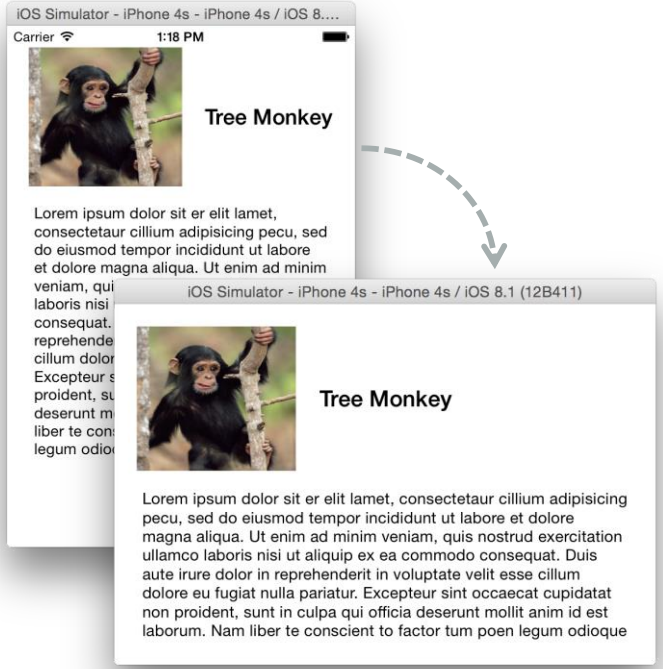
Tasks

1. Use constraint inequalities and priorities to adaptively place views
2. Recognize constraint issues in the designer



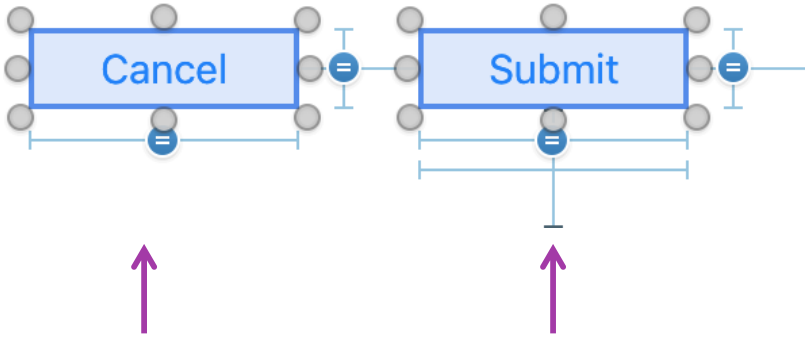
Benefits of Auto Layout

- Auto Layout allows us to create UIs that:
  - ✓ React to orientation changes
  - ✓ React to parent size changes
  - ✓ Handle dynamic content (user selectable fonts, localization, etc.)
  - ✓ Support multiple devices / form factors with a single UI
  - ✓ Universal Storyboards



Reminder: What are Constraints?

- Auto Layout uses *constraints* to decide the position and size of each view



Constraints are applied between views to align, size, and space them relative to each other

Decided by constraints

- Leading or Trailing
- Top or bottom
- Width
- Height
- Alignment

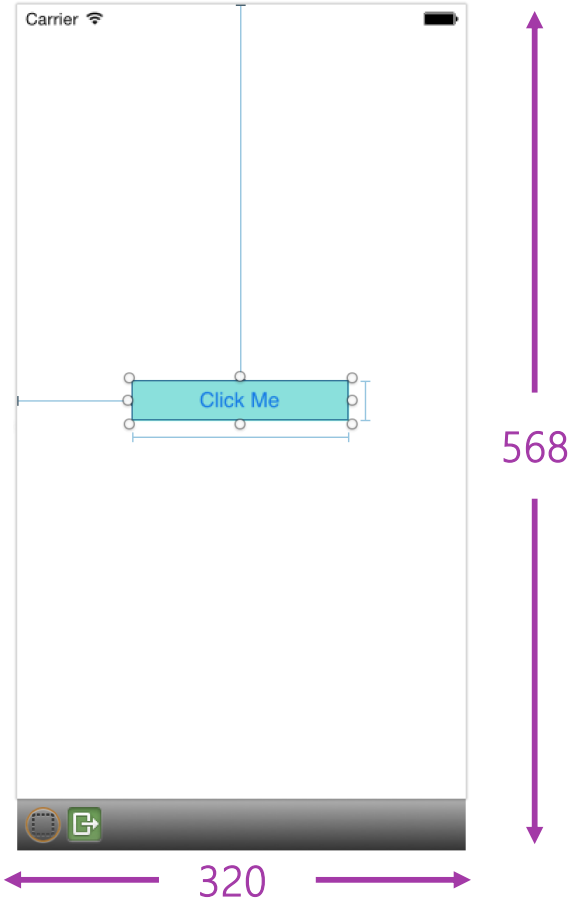
Flash Quiz

Xamarin University

Flash Quiz

① Q: Given the following constraints, **what will the size and position of the button be when this device is rotated to landscape?**

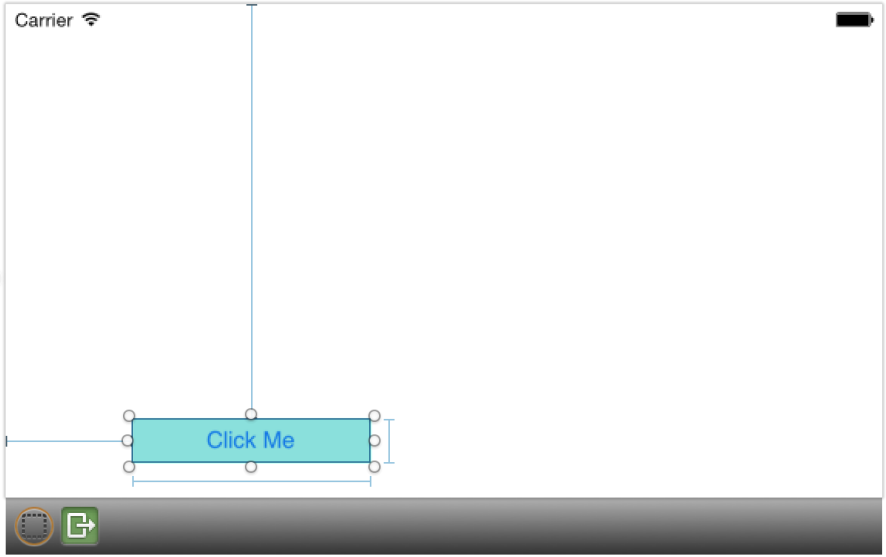
Constraint: left edge	= 80
Constraint: top edge	= 250
Constraint: width	= 160
Constraint: height	= 40



Flash Quiz

① A: The size of the button is unchanged, and the spacing from the left and top edges will be the same

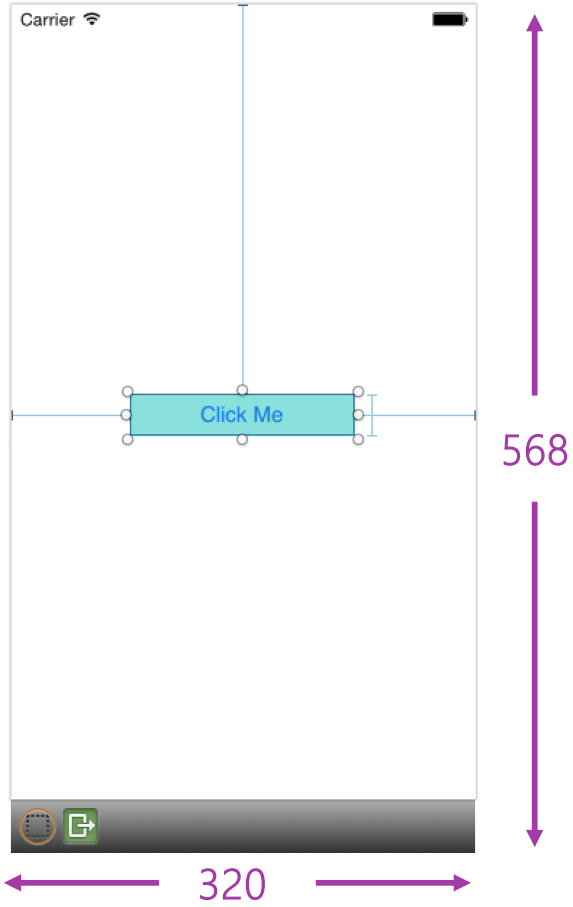
Constraint: left edge = 80
Constraint: top edge = 250
Constraint: width = 160
Constraint: height = 40



Flash Quiz

② Q. Given the following constraints, **what will the size and position of the button be when this device is rotated to landscape?**

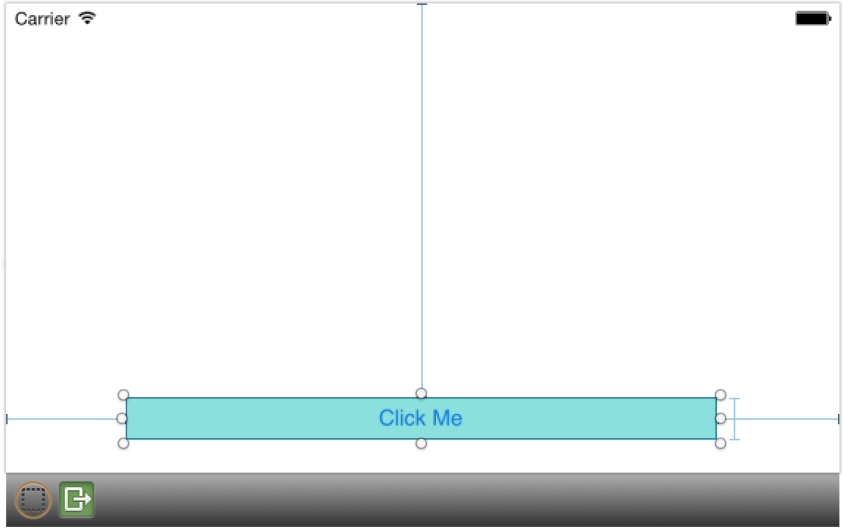
Constraint: left edge = 80
Constraint: right edge = 80
Constraint: top edge = 250
Constraint: height = 40



Flash Quiz

② A. The button is stretched according to the left and right constraints.

Constraint: left edge = 80
Constraint: top edge = 250
Constraint: right edge = 80
Constraint: height = 40



What is a Constraint?

- Each constraint is a linear equation with the following format:

```
view1.attribute = multiplier * view2.attribute + constant
```

Constraint first attribute

- Constraints ultimately calculate a single value or *attribute* for a view that will be used to calculate size or position

```
view1.attribute = multiplier * view2.attribute + constant
```

Attribute specifies the value we want to constrain – e.g. the position, alignment or size

MonoTouch.UIKit.NSLayoutAttribute

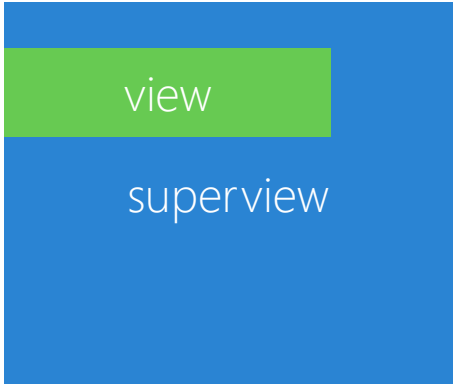
- F Baseline
- F Bottom
- F BottomMargin
- F CenterX
- F CenterXWithinMargins
- F CenterY

Constraint second attribute

- Often, the attribute to be calculated is based on an attribute from another a view

```
view1.attribute = multiplier * view2.attribute + constant
```

Assigned value *typically* comes from an attribute on a sibling or ancestor view




```
view.left = 1 * superview.left + 0
```

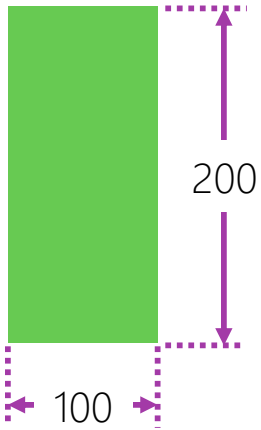
Constraint multiplier

- The *multiplier* is used to scale the input attribute

```
view1.attribute = multiplier * view2.attribute + constant
```

Multiplier allows value to be scaled – e.g. *half* or *double* the source value and cannot be zero

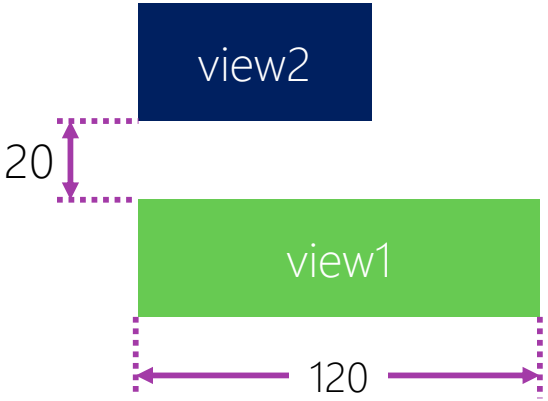
```
view.height = 2 * view.width + 0
```



Constraint constant

- A numerical value can be added to final value by assigning a *constant*

```
view1.attribute = multiplier * view2.attribute + constant
```



Constant allows a specific value to be added or subtracted from the equation, defaults to 0

```
view1.top = 1 * view2.bottom + 20
```

```
view1.width = 120
```

Constraint equality

- Constraints are linear equations that may require additional constraints to solve the final single attribute value

```
view1.attribute >= multiplier * view2.attribute + constant
```

Can be **=**, **>=** or **<=**

Can use inequalities to allow for min/max values to be applied

```
view1.width <= 0.5 * superview.width
```

Max value for **view1**

```
view1.width >= 250
```

Min value for **view1**

Intrinsic Content Size

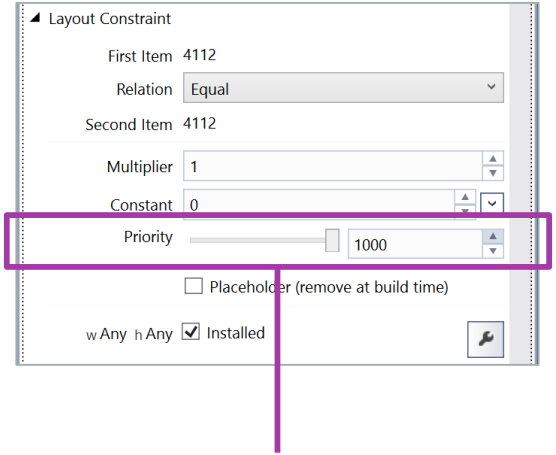
- Some **`UIViews`** can calculate their preferred size based on their content - a constraint to decide the width and/or height is not required

- **`UILabel`** and **`UIButton`** sizes to the text content
- **`UISwitch`** has a default size
- **`UIImageView`** sizes to the image

Can apply constraints to override intrinsic sizing

Constraint priority

- The *Priority* of a constraint determines which constraint is used to determine layout when there's a conflict



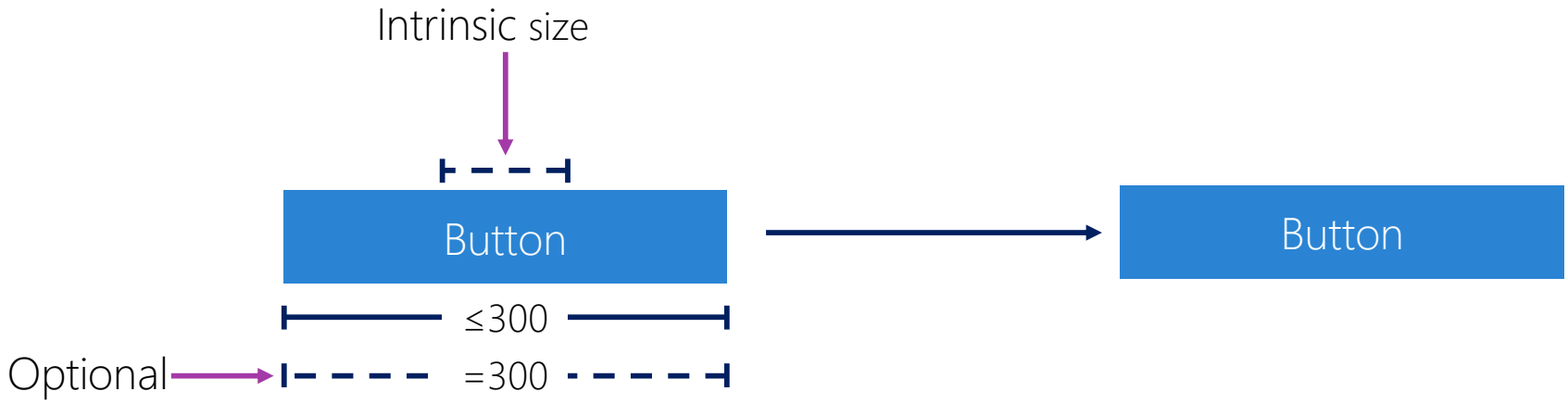
Priority values are between 0 and 1000 with 1000 meaning "required"

Adaptive UI

- Constraints defined with inequalities rely on other constraints to find the a final value

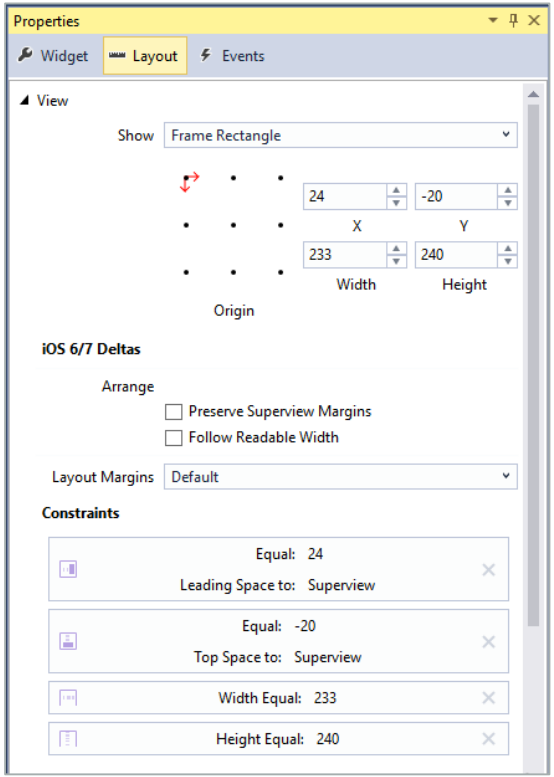
Adaptive sizing

- We can define *optional* or lower-priority constraints to tell Auto Layout our preferred value



Properties Pane

- The Xamarin iOS Designer includes a properties pane that allows you to adjust and configure all aspects of design-defined constraints



Demonstration

Constraint equality and priorities

Xamarin University

Common layout problems

- Three common problems can cause our views to not be positioned or displayed correctly at runtime

Not enough constraints

Apple refers to this as an *ambiguous layout* and it results in views being positioned incorrectly, or not shown

Common layout problems

- Three common problems can cause our views to not be positioned or displayed correctly at runtime

Not enough constraints

Conflicting constraints

This happens when multiple constraints affect the same property of a view but do not agree on the value

Common layout problems

- Three common problems can cause our views to not be positioned or displayed correctly at runtime

Not enough constraints

Conflicting constraints

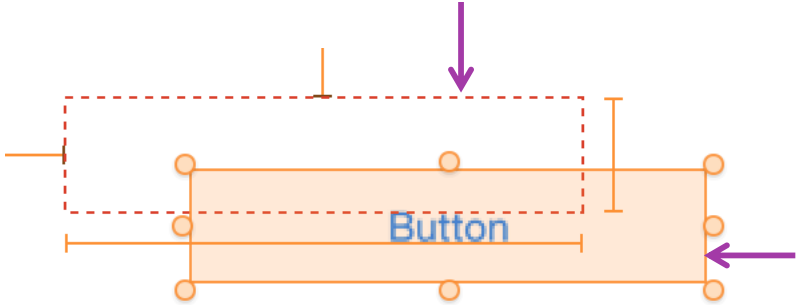
Misplaced views

Design-time layout does not match runtime layout

Runtime constraint issues

- The runtime location and size of your views may differ from the designer representation

The dotted outline indicates the real (runtime) position



The orange highlight indicates the frame (design time) position

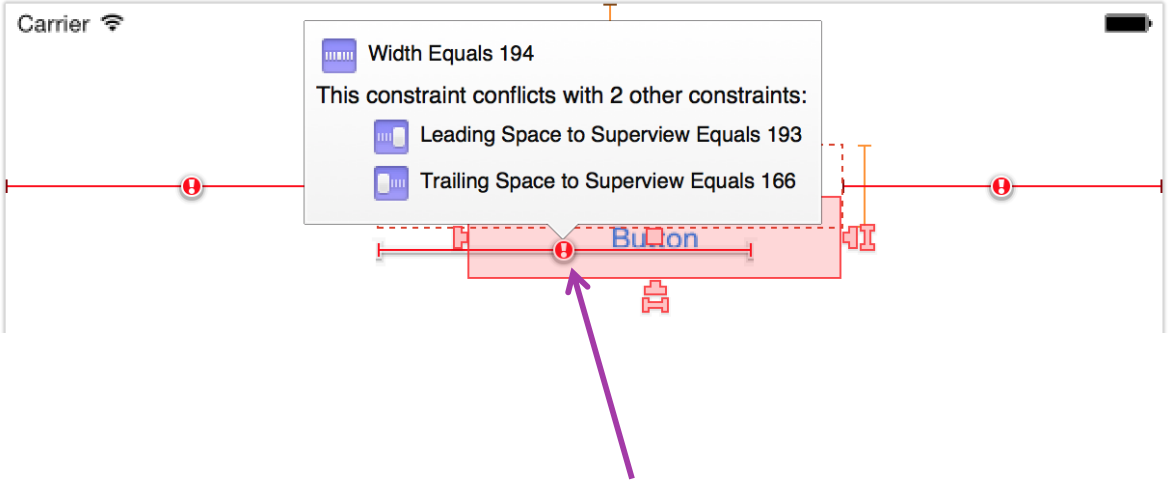
Coordinating design and runtime layout

- The designer provides controls to help synchronize design-time and runtime layout

Ask the designer to update the **frame** to match the current constraints

Conflicting Constraints

- Conflicting constraints are marked in red and show a warning symbol



Hovering over the warning symbol shows additional information

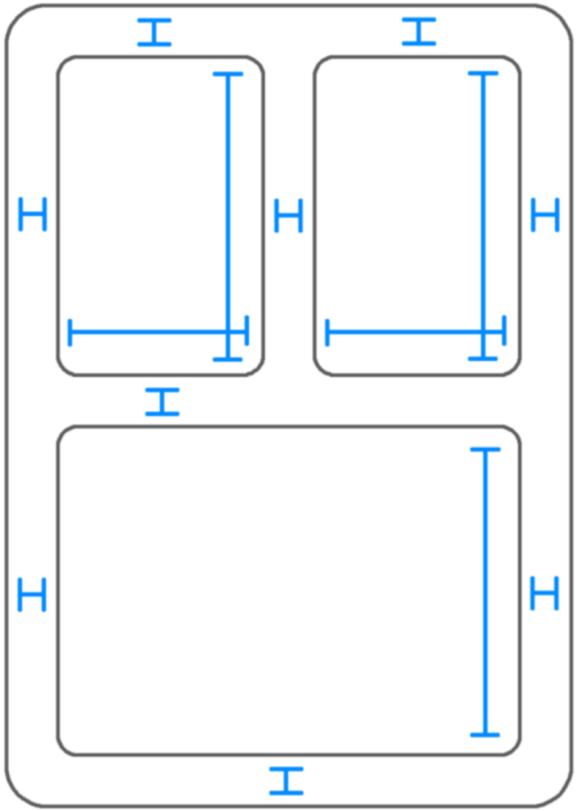
Individual Exercise

Use Auto Layout in the Designer

Xamarin University

Summary

1. Use constraint inequalities and priorities to adaptively place views
2. Recognize constraint issues in the designer



Create and update constraints programmatically

Xamarin University

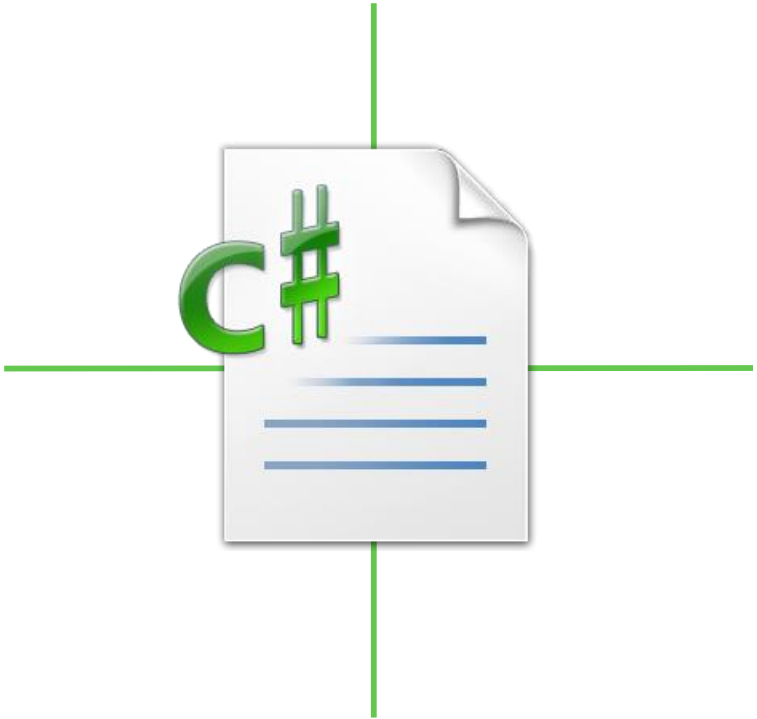
Tasks

- Create constraints in code
- Update constraints at runtime

```
is.View.RemoveConstraint (constLeftText);
is.View.RemoveConstraint (constTopText);

/create new constraints based on orientation
f (isLandscape == true)
{
    // Going landscape. Move text to the righ
    constLeftText = GetConstraint (constraint
        // Text left constraint now attaches
        object2: imgMonkey,
        // ...and constrains to the monkey im
        attribute2: NSLayoutAttribute.Trailin
    );

    constTopText = GetConstraint (constraint:
        // Top edge constraint of the text no
        object2: this.View,
        // ...and to top layout attribute ins
        attribute2: NSLayoutAttribute.Top
    );
}
else
{
    constLeftText = GetConstraint (constraint
        // Text left constraint new attaches
        object2: this.View,
        // ...and the left edge of the parent
        attribute2: NSLayoutAttribute.Leading
    );

    constTopText = GetConstraint (constraint:
        // Constrain the text to the monkey a
        object2: imgMonkey,
        // ...and use the bottom edge of the
        attribute2: NSLayoutAttribute.Bottom
```

Advantages of code-based layout

- Can create and organize dynamic content / controls in our views
- Can update constraints based on runtime changes such as orientation
- Can provide layout animations to make your app look professional



Auto Layout APIs

- Auto Layout support uses **`NSLayoutContraint`** to create and manage constraints

```
public class NSLayoutConstraint : NSObject
{
    public float Constant { get; set; }
    public float Priority { get; set; }
    ...
    public static NSLayoutConstraint.Create(...)
}
```

Static factory methods used to create constraints

Creating Constraints

- The static **`NSLayoutConstraint.Create`** method is used to create new constraints programmatically

```
NSLayoutConstraint newConstraint = NSLayoutConstraint.Create
(
    firstView, view1attribute,
    relationship,
    secondView, view1attribute,
    multiplier, constant
);
```

Constraint objects

- Every constraint references at least one object or view; typically a constraint will identify two views and define a relationship between them

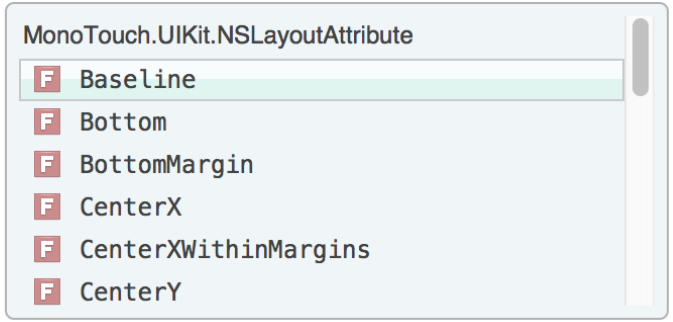
```
NSLayoutConstraint newConstraint = NSLayoutConstraint.Create
(
    firstView, view1attribute,
    relationship,
    secondView, view1attribute,
    multiplier, constant
);
```

The second object can be **null** if we're defining a constant value to a constraint

Attributes in Constraints

- Constraint attributes determine which layout property of the view is used in the constraint

Attributes are defined by the **NSLayoutAttribute** enumeration

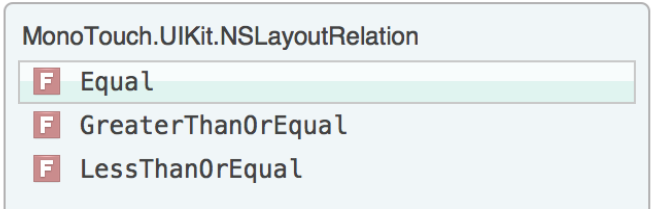



```
var newConstraint = NSLayoutConstraint.Create (
    firstView, view1attribute,
    relationship,
    secondView, view2attribute,
    multiplier, constant);
```

Constraint Relationship

- The relationship between two view properties in the constraint can be **`Equal`**, **`GreaterThanOrEqual`**, or **`LessThanOrEqual`**

Relationships are defined by the **`NSLayoutRelation`** enumeration




```
var newConstraint = NSLayoutConstraint.Create (
    firstView, view1attribute,
    relationship,
    secondView, view2attribute,
    multiplier, constant);
```

UIView Constraint Methods

- The **`UIView`** base class has a number of methods and properties for interacting with constraints

```
var constraints = myView.Constraints;
```

Get the constraint array for the view

```
myView.AddConstraint (newConstraint);
myView.AddConstraints (constraints);
```

Add one or more constraints

```
myView.RemoveConstraint (constraint);
myView.RemoveConstraints (constraints);
```

Remove one or more constraints

```
myView.SetNeedsUpdateConstraints ();
```

Update view layout

Disable autoresizing mask

- Set **`TranslatesAutoresizingMaskIntoConstraints`** to **`false`** to prevent the system from automatically creating a set of constraints

```
var myButton = new UIButton(...);
myButton.TranslatesAutoresizingMaskIntoConstraints = false;

NSLayoutConstraint leftConstraint = NSLayoutConstraint.Create(
    myButton, NSLayoutAttribute.Left, NSLayoutRelation.Equal,
    this.View, NSLayoutAttribute.Left, 1 ,
...
this.View.AddConstraint (leftConstraint);
```

Turn off "springs and struts" support for the view you are working with

Create a constraint in code

- Create an **NSLayoutConstraint** using the **Create** method

```
var myButton = new UIButton(...);
myButton.TranslatesAutoresizingMaskIntoConstraints = false;

NSLayoutConstraint leftConstraint = NSLayoutConstraint.Create(
    myButton, NSLayoutAttribute.Left, NSLayoutRelation.Equal,
    this.View, NSLayoutAttribute.Left, 1 , 150);
...
this.View.AddConstraint (leftConstraint);
```

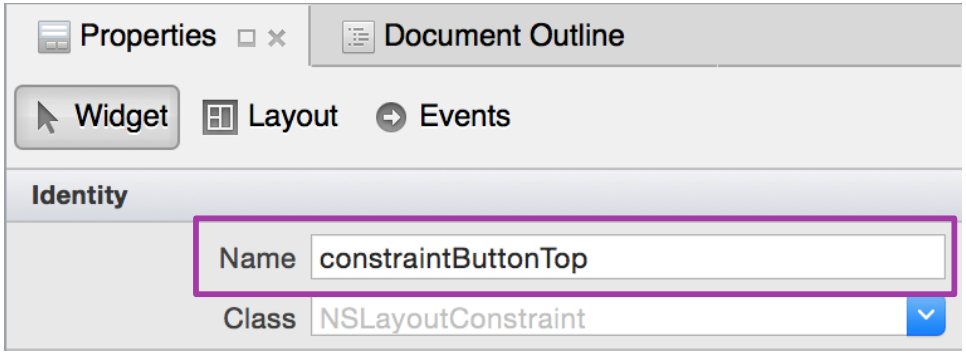
Add Constraints to a View

- To add a constraint, we use the **`UIView`** methods **`AddConstraint`** or **`AddConstraints`** (when adding a collection of constraints)

```
var myButton = new UIButton(...);
myButton.TranslatesAutoresizingMaskIntoConstraints = false;

NSLayoutConstraint leftConstraint = NSLayoutConstraint.Create(
    myButton, NSLayoutAttribute.Left, NSLayoutRelation.Equal,
    this.View, NSLayoutAttribute.Left, 1 , 150);
...
this.View.AddConstraint (leftConstraint);
```

Naming Constraints in the Designer

- Setting a name for a constraint created in the designer creates a property allowing us to interact with the constraint programmatically




```
constraintButtonTop.Constant = 100;
```

Individual Exercise

Update Constraints Programmatically

Xamarin University

Flash Quiz

Xamarin University

Flash Quiz

① How many constraints are required to fully constrain a view?

a) 1

b) 2

c) 4

d) More than 4

Flash Quiz

① How many constraints are required to fully constrain a view?

a) 1

b) 2

c) 4* (unless relying on intrinsic content size)

d) More than 4

Flash Quiz

② When setting a constraint value to be exactly equal to another attribute, what value do we assign to the constant?

a) 0
b) 1
c) 2
d) Any value, it doesn't matter

Flash Quiz

② When setting a constraint value to be exactly equal to another attribute, what value do we assign to the constant?

a) 0
b) 1
c) 2
d) Any value, it doesn't matter

Summary

- Create constraints in code
- Update constraints at runtime

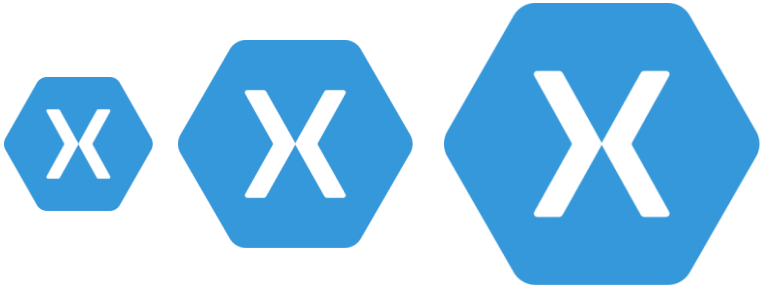
```
is.View.RemoveConstraint (constLeftText);
is.View.RemoveConstraint (constTopText);

/create new constraints based on orientation
f (isLandscape == true)
{
    // Going landscape. Move text to the righ
    constLeftText = GetConstraint (constraint
        // Text left constraint now attaches
        object2: imgMonkey,
        // ...and constrains to the monkey im
        attribute2: NSLayoutAttribute.Trailin
    );

    constTopText = GetConstraint (constraint:
        // Top edge constraint of the text no
        object2: this.View,
        // ...and to top layout attribute ins
        attribute2: NSLayoutAttribute.Top
    );
}
else
{
    constLeftText = GetConstraint (constraint
        // Text left constraint new attaches
        object2: this.View,
        // ...and the left edge of the parent
        attribute2: NSLayoutAttribute.Leading
    );

    constTopText = GetConstraint (constraint:
        // Constrain the text to the monkey a
        object2: imgMonkey,
        // ...and use the bottom edge of the
        attribute2: NSLayoutAttribute.Bottom
```

Animate constraint changes

Xamarin University

Tasks

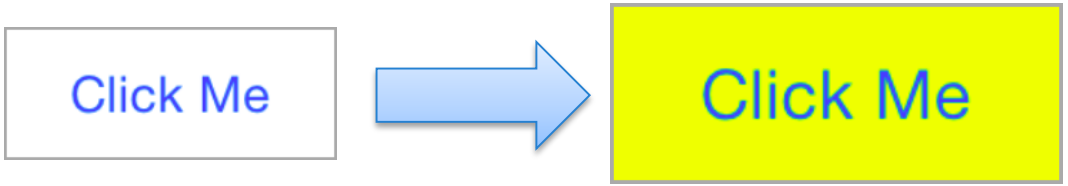
1. Animate view property changes
2. Animate constraint changes



Animating Property Changes

- You apply **UIView** animations to most view properties using the static **Animate** method

```
UIView.Animate (duration: 0.5, animation: () => {
    myButton.BackgroundColor = UIColor.Yellow;
    myButton.Transform =
        CGAffineTransform.MakeScale(1.5f, 1.5f);
};
```



Constraint Animations

- **`UIView`** animations can be used to animate changes when updating constraint properties or adding/removing constraints

`AddConstraint` and **`RemoveConstraint`** are called on the View

```
this.RemoveConstraint (constraintOld);
this.AddConstraint (constraintNew);

UIView.Animate (duration: 0.5, animation: () => {
    this.View.LayoutIfNeeded();
};
```

Call **`LayoutIfNeeded`** on the parent to force a layout recalculation during the animation

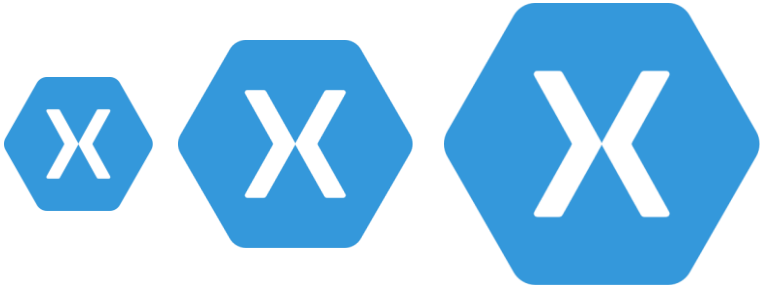
Exercise

Animate Constraints

Xamarin University

Summary

1. Animate view property changes
2. Animate constraint changes

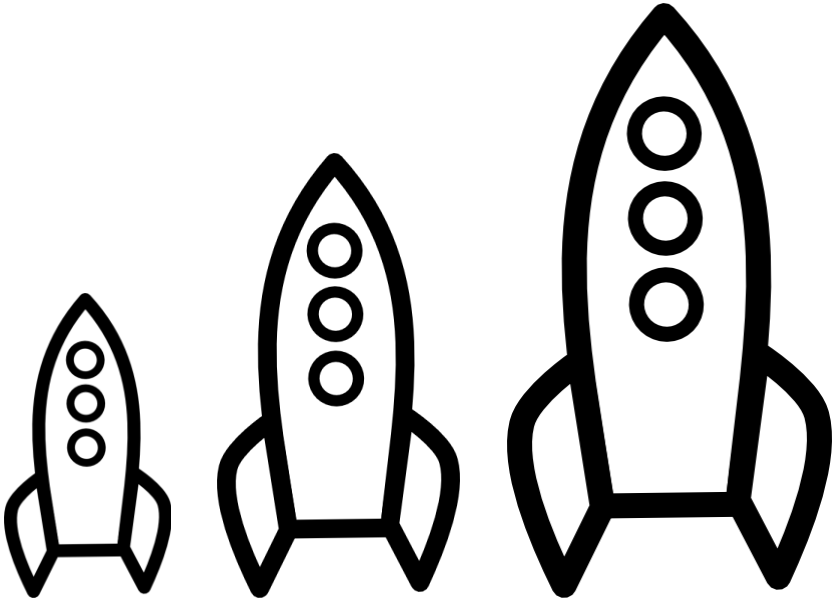


Use Size Classes to customize your UI for different screen sizes

Xamarin University

Tasks

1. Change the size class in the designer
2. Customize the UI for different screen sizes

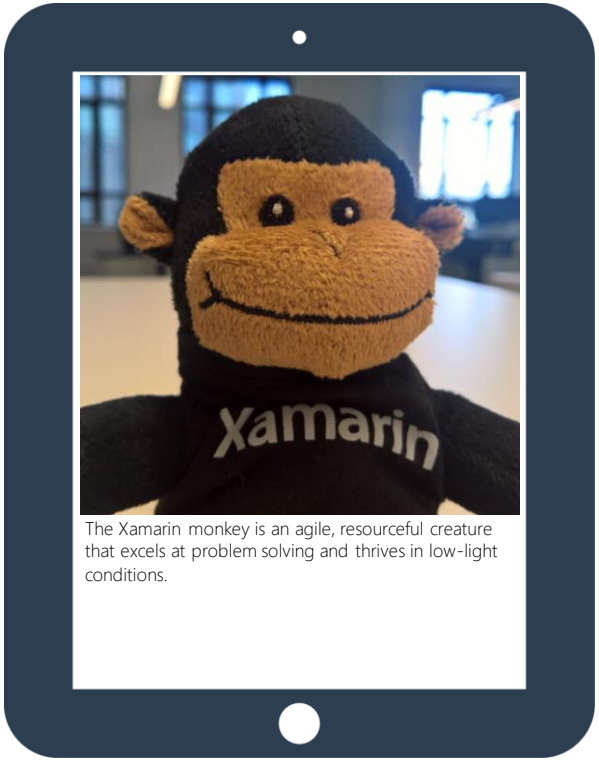


Cross-device UI is challenging

- A responsive UI doesn't automatically produce an ideal user experience across different screen sizes and orientations

Proportions looks great on a phone

Image is too large and the font is small



What are size classes?

- Size classes are a categorization of physical or virtual width and height of a UI element – most commonly used to categorize screen size

Size Class Definitions

- Content area is determined by how much space is available horizontally and vertically - each dimension can be one of two values:

Indicates that the specified dimension has more space available

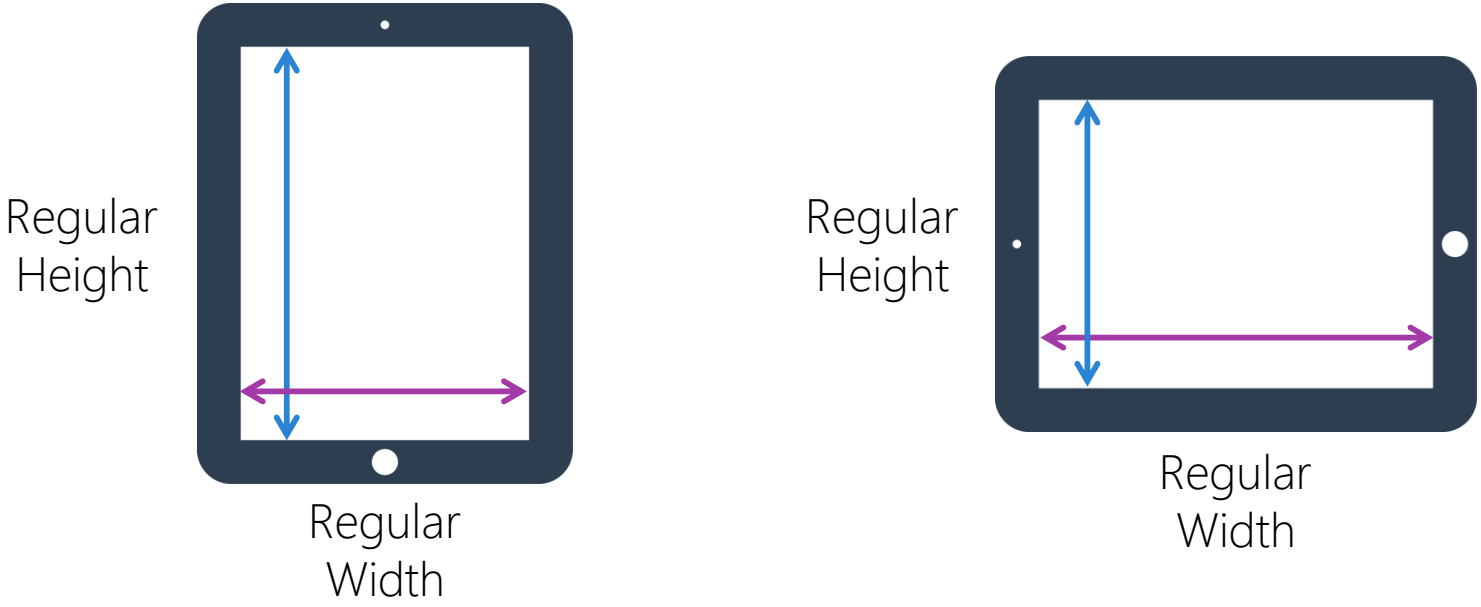
Regular

Compact

Indicates that the dimension is smaller and has limited viewing capability

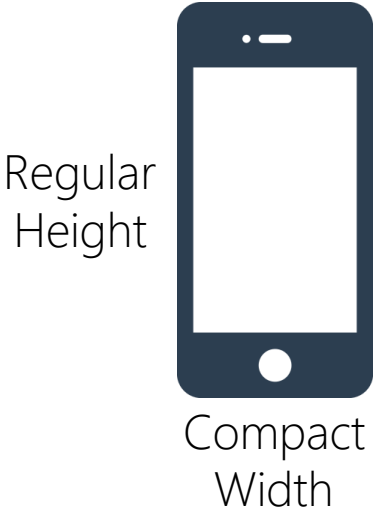
iPad Size Classes

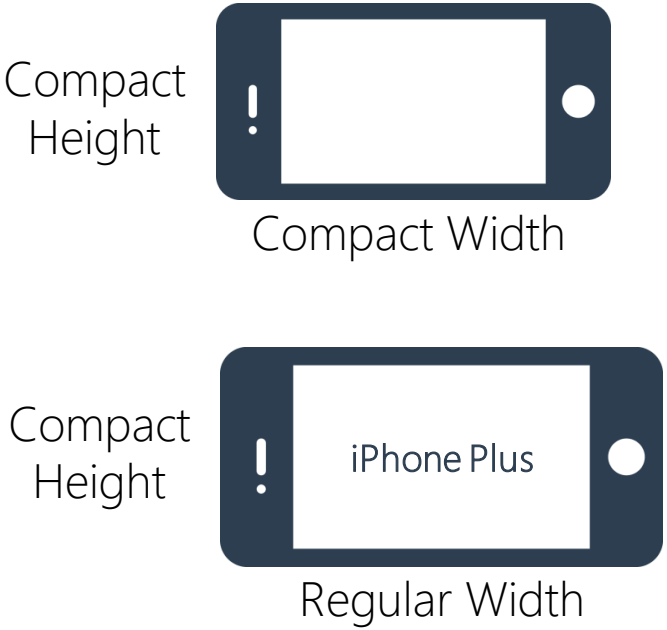
- iPads have large screens are considered to have regular width and regular height regardless of their orientation – or a *Regular-Regular Size Class*



iPhone Size Classes

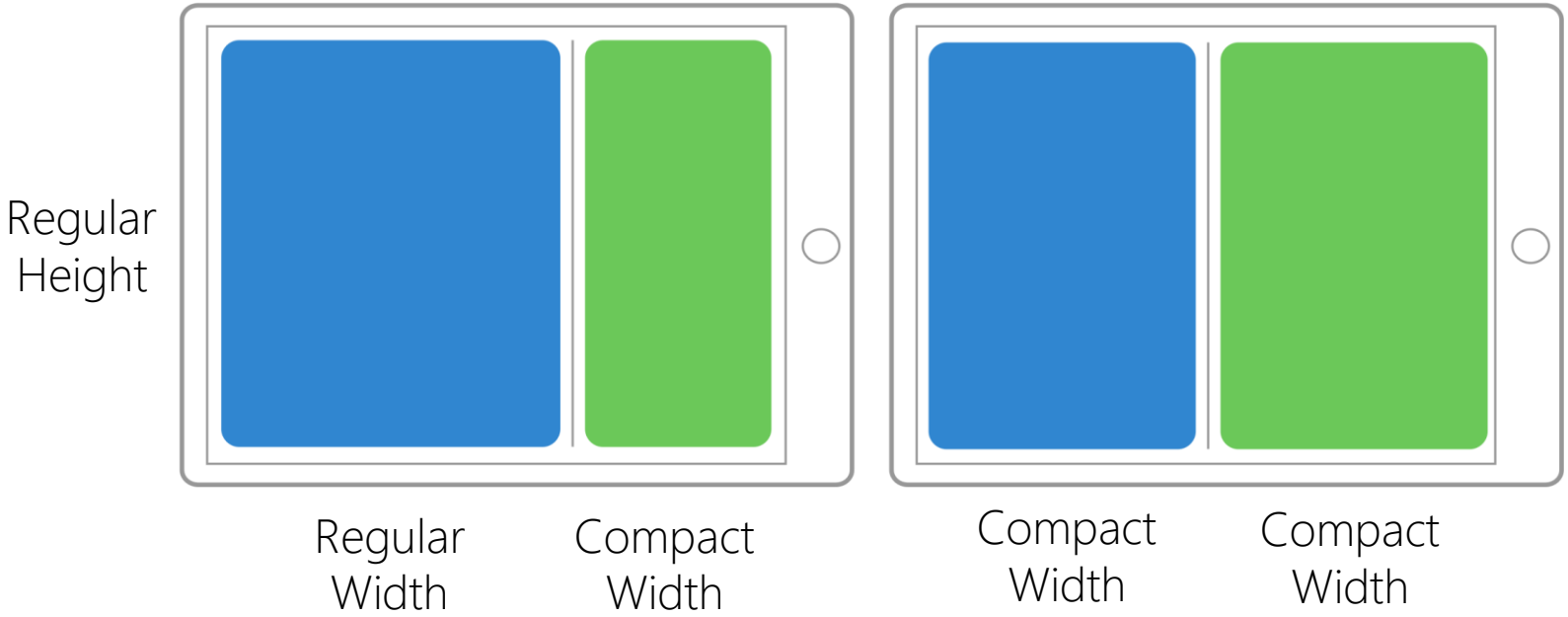
- iPhone size classes change with orientation and device size





Size Classes and the Split View

- The size class can change for applications shown within a split view on the iPad



Size Classes in the Designer

- The Xamarin.iOS Designer allows you to view your UI as it would appear on real devices and indicates the related size classes

wR indicates horizontal size class: Width Regular
hC indicates vertical size class: Height Compact

Demo

Changing the Size Class in the Xamarin.iOS Designer

Xamarin University

Customizing per Size Class

- The iOS Designer allows the UI to be customized for each Size Class

Add / remove Constraints

Change constraint properties

Add / remove Views

Change Fonts

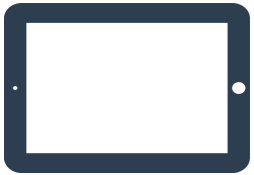
Designer Size Class

- The iOS Designer recognizes 9 different size classes – four Final Size Classes and five Base Size Classes

Compact Width Compact Height	Any Width Compact Height	Regular Width Compact Height
Compact Width Any Height	Any Width Any Height	Regular Width Any Height
Compact Width Regular Height	Any Width Regular Height	Regular Width Regular Height

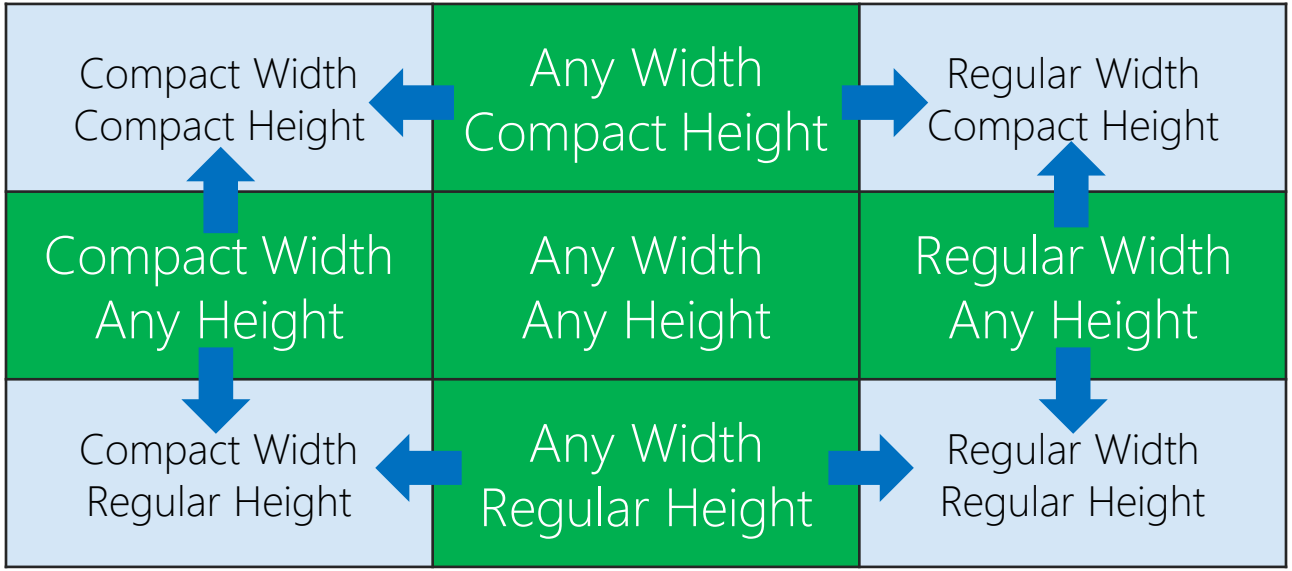
Final Size Classes

- Final Size Classes represent size classes for real devices

Compact Width Compact Height 	Regular Width Compact Height 
Compact Width Regular Height 	Regular Width Regular Height 

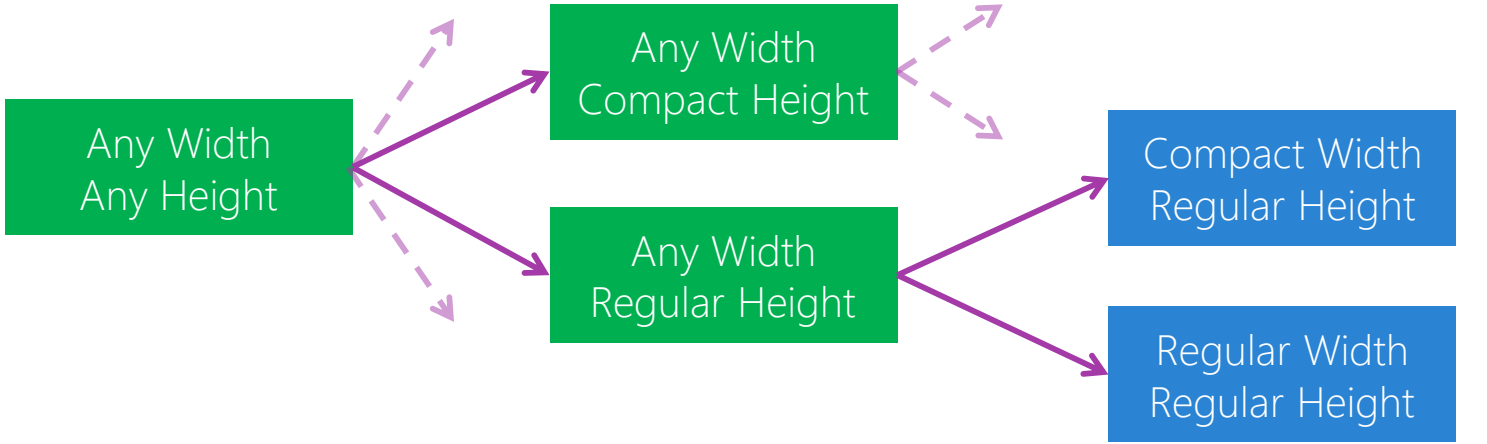
Base Size Classes

- Base Size Classes are abstract size classes that represent two or more Final Size classes - visualized as square devices in the iOS Designer



Base Size Class changes

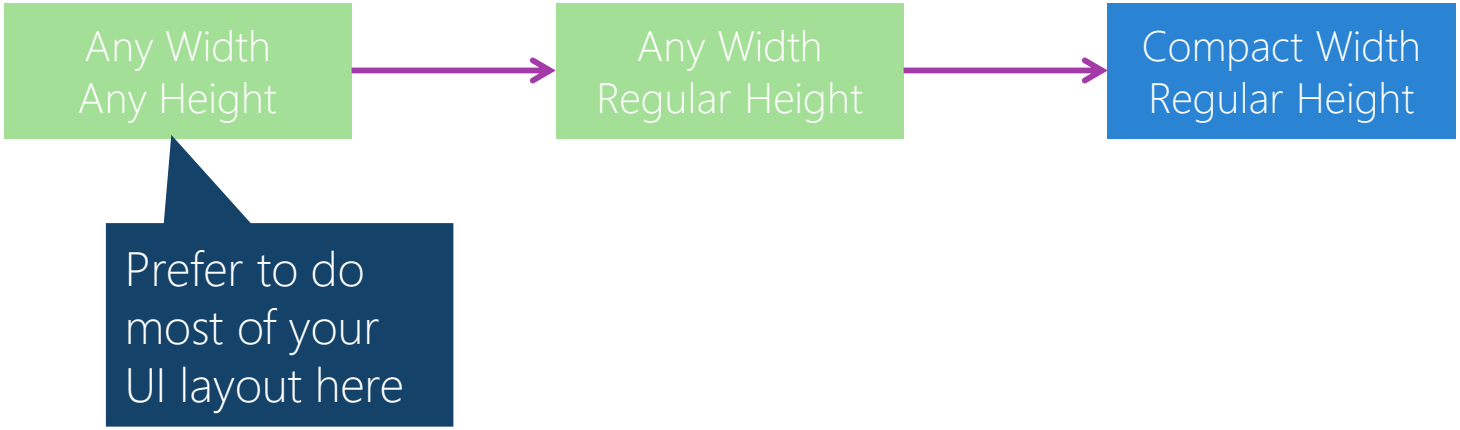
- Changes made in the base size classes will be reflected in the corresponding final size classes



Changes propagate from generic to specific

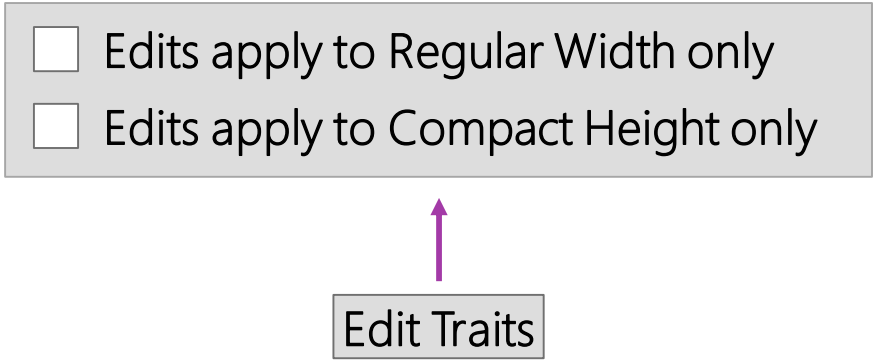
Size Class Hierarchy

- Changes made in a final size class will override changes in the corresponding base size classes for that final size class only



Editing for a final size class

- You can make edits to a final size class by selecting a device/orientation combination that matches your desired size class and then press the **Edit Traits** button



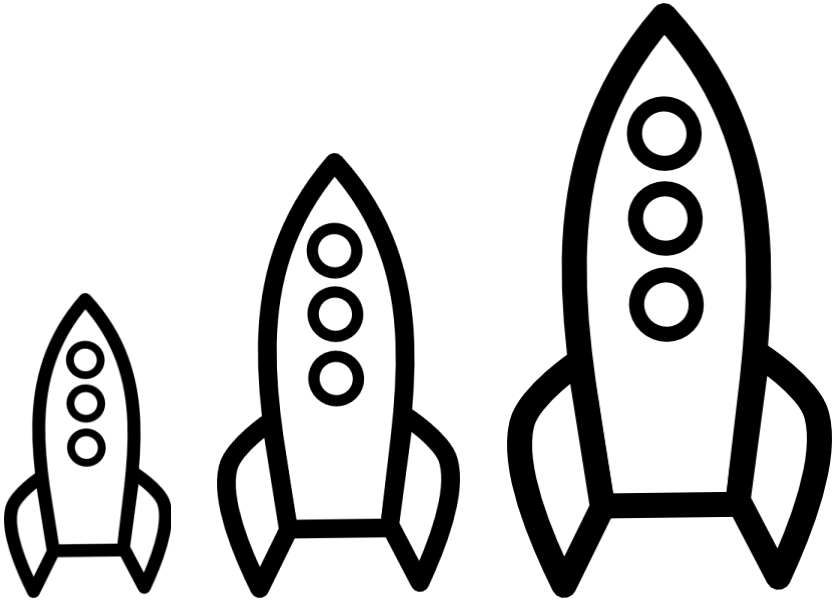
Exercise

Update your UI based on the Size Class

Xamarin University

Summary

1. Change the size class in the designer
2. Customize the UI for different screen sizes



Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

Microsoft