Thinking In NSLayoutConstraints

Who Am I?

- I run M Cubed Software (mcubedsw.com)
- Built many apps using Auto Layout
- Last year I talked about how Auto Layout thinks
- This year I'll talk about how you should think

What Is Auto Layout?

- Constraint-based layout system for iOS & Mac
- Define relationships between views
- Introduced in Mac OS X 10.7 and iOS 6
- Make previously complex layout problems simple
- Requires a different way of thinking about layout
- Fits more closely to your natural mental model

Constraints: How Do They Work?

Constraints

- Represented by NSLayoutConstraint
- Defines relationship between two attributes
- Attributes are effectively variables
- Treat a constraint as small function modifying a variable

```
y = mx + c
```

CONSTRAINTS

- Represented by NSLayoutConstraint
- Defines relationship between two attributes
- Attributes are effectively variables
- Treat a constraint as small function modifying a variable

```
view1.attribute = m * x + c
```

Constraints

- Represented by NSLayoutConstraint
- Defines relationship between two attributes
- Attributes are effectively variables
- Treat a constraint as small function modifying a variable

```
view1.attribute = m * view2.attribute + c
```

Constraints

- Represented by NSLayoutConstraint
- Defines relationship between two attributes
- Attributes are effectively variables
- Treat a constraint as small function modifying a variable

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

Constraints

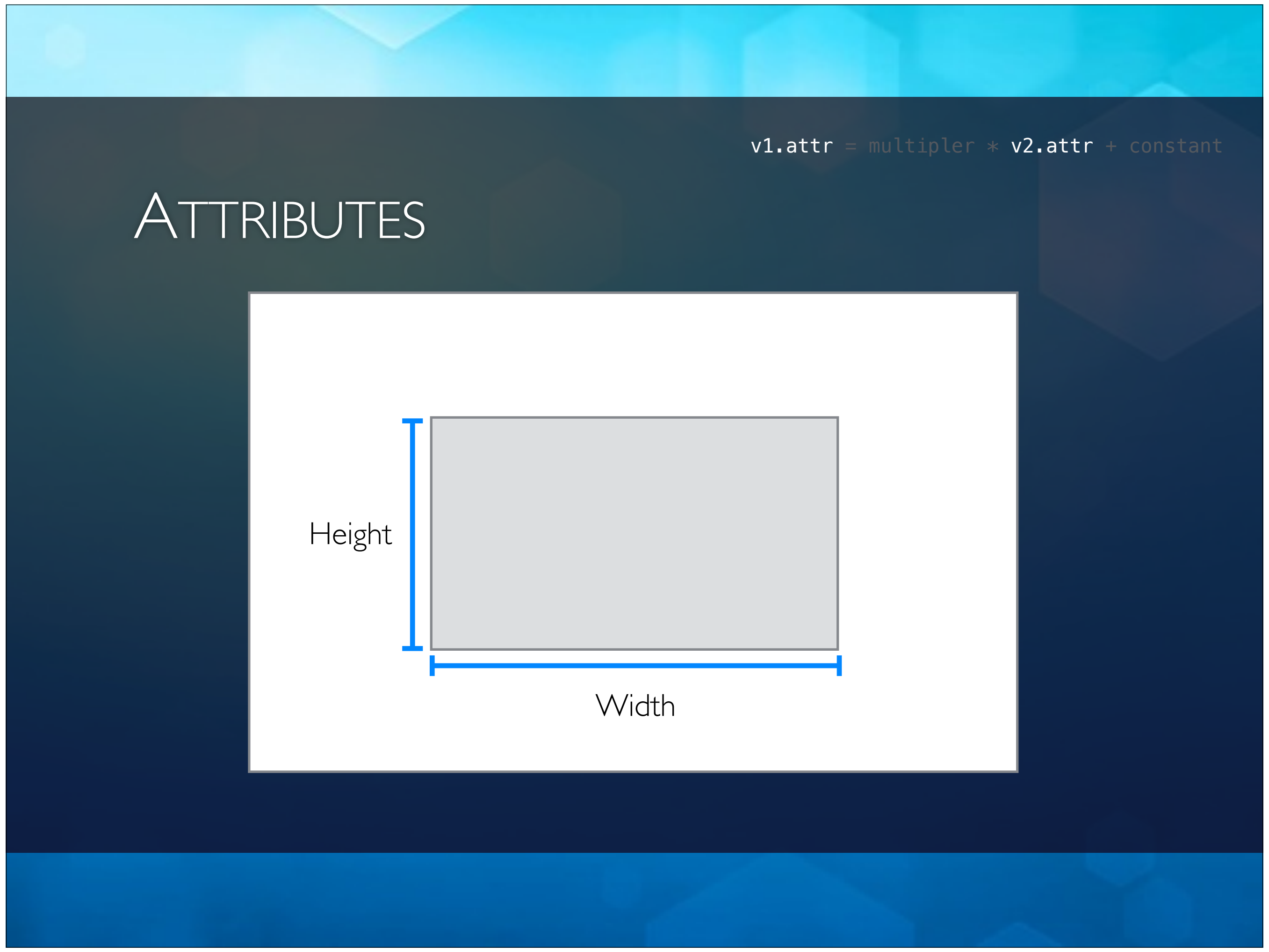
- Represented by NSLayoutConstraint
- Defines relationship between two attributes
- Attributes are effectively variables
- Treat a constraint as small function modifying a variable

```
v1.attr = multipler * v2.attr + constant
```

```
v1.attr = multipler * v2.attr + constant
```

CONSTRAINTS

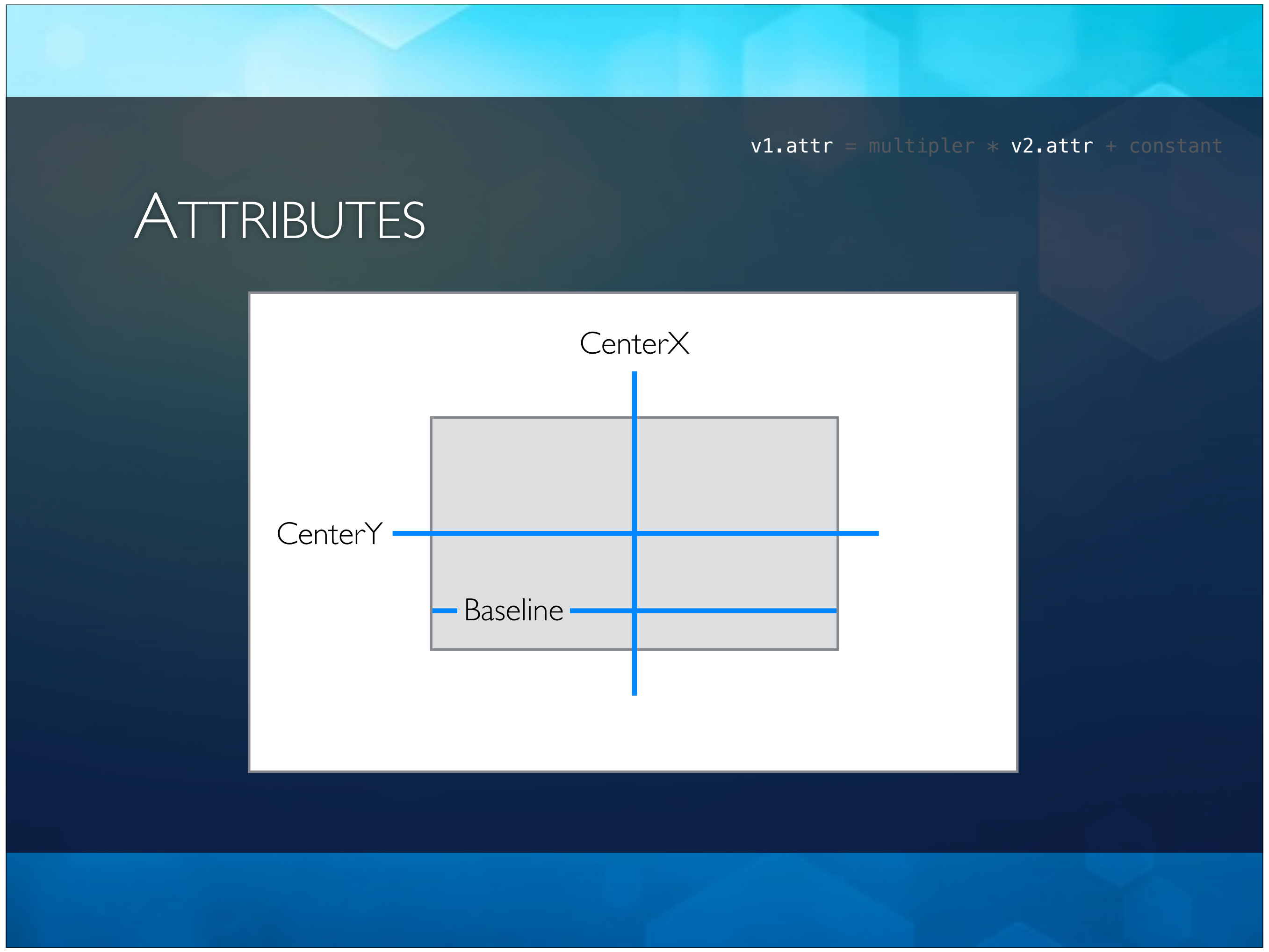
- Represented by NSLayoutConstraint
- Defines relationship between two attributes
- Attributes are effectively variables
- Treat a constraint as small function modifying a variable



```
v1.attr = multipler * v2.attr + constant
```

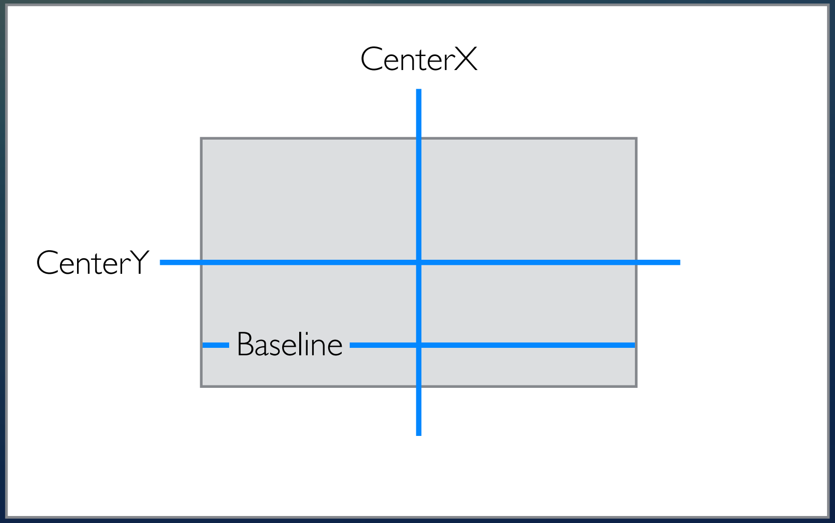
ATTRIBUTES

Attributes are available for height and width

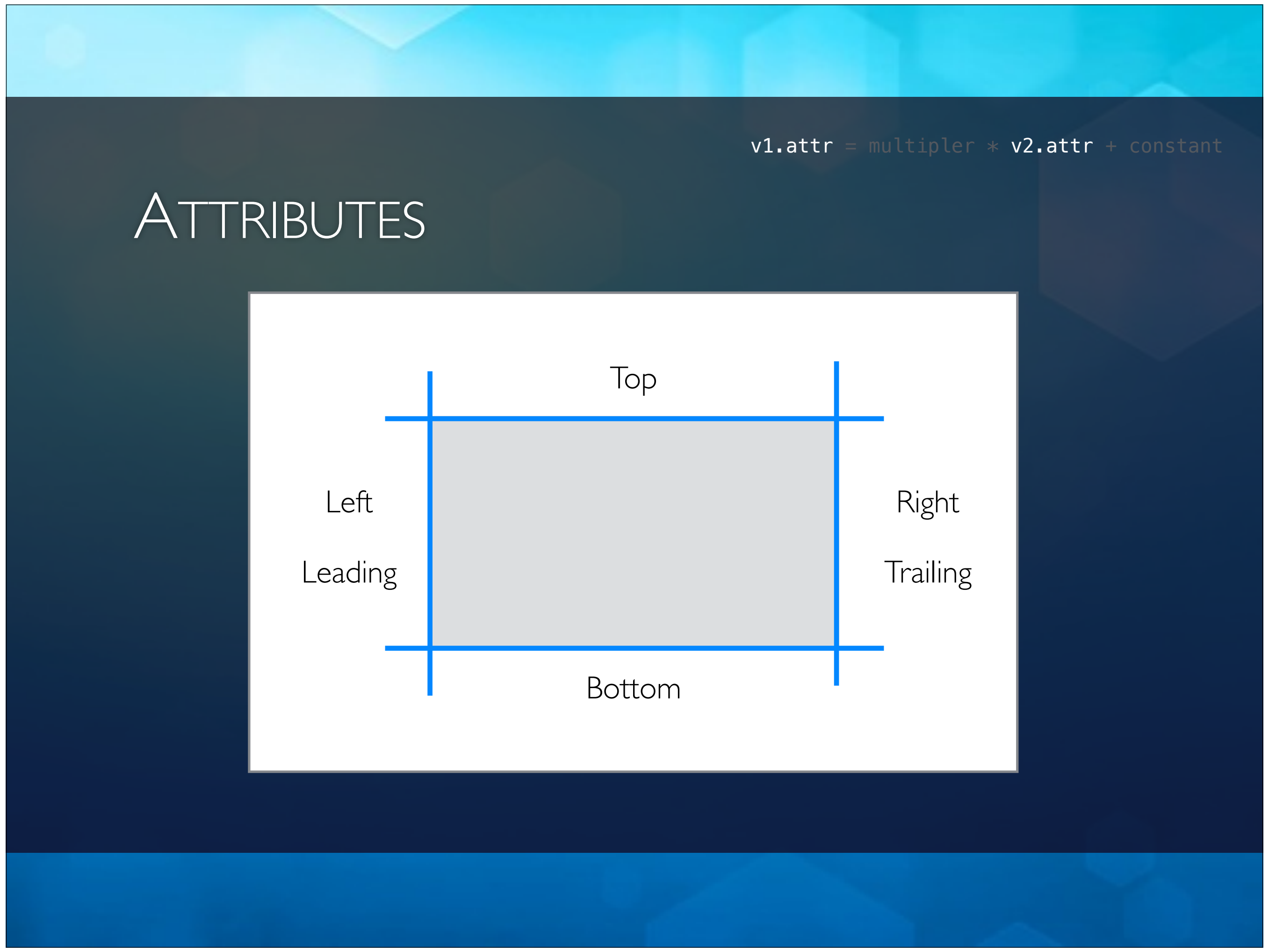


`v1.attr = multipler * v2.attr + constant`

ATTRIBUTES

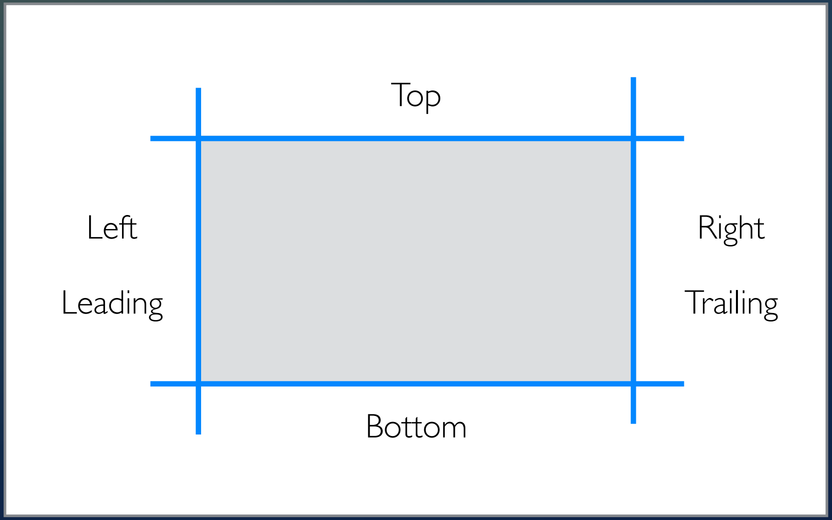


And for horizontal centre (Center X), vertical centre (Center Y) and baseline

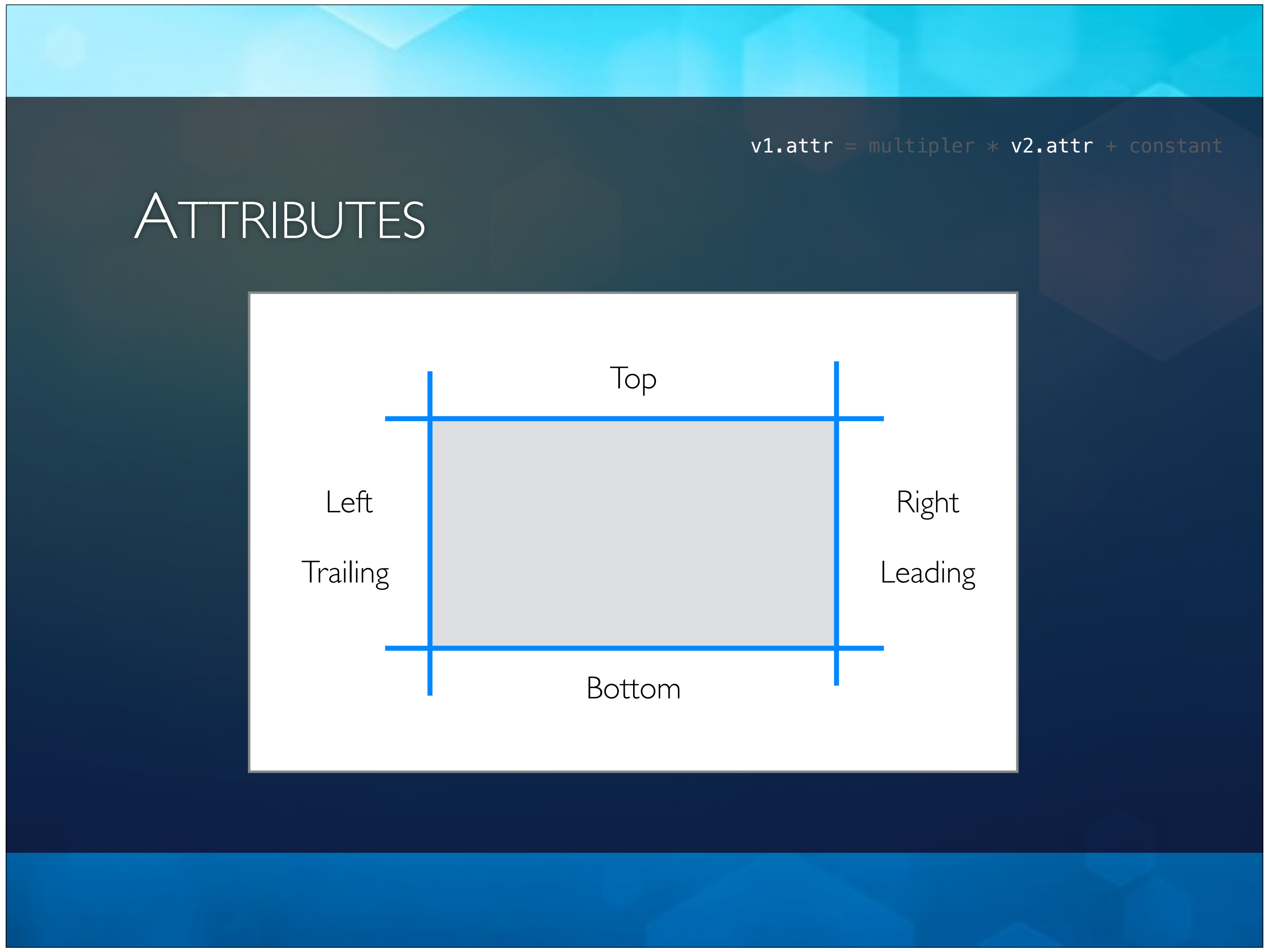


`v1.attr = multipler * v2.attr + constant`

ATTRIBUTES

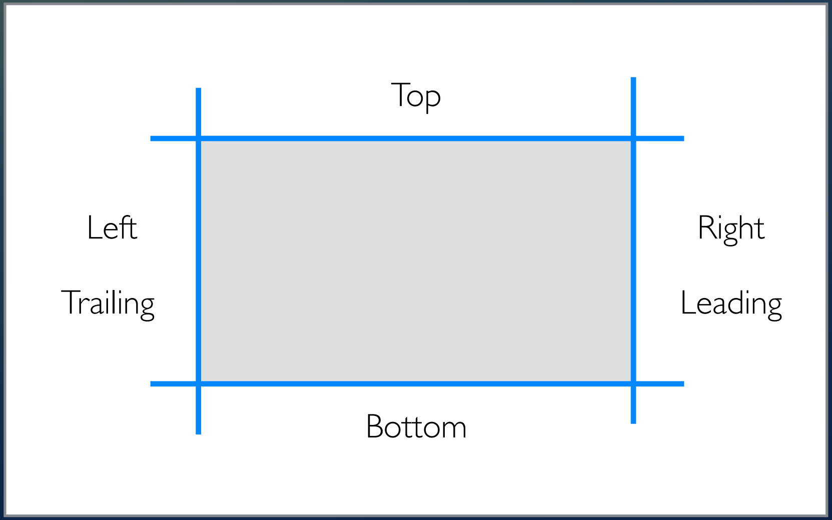


Also for left/leading, right/trailing, top and bottom



v1.attr = multipler * v2.attr + constant

Attributes



In a right to left language, trailing and leading swap round. If you are using these rather than left and right, then your UI will flip around after you provide a localisation, saving you a lot of work

```
v1.attr = multipler * v2.attr + constant
```

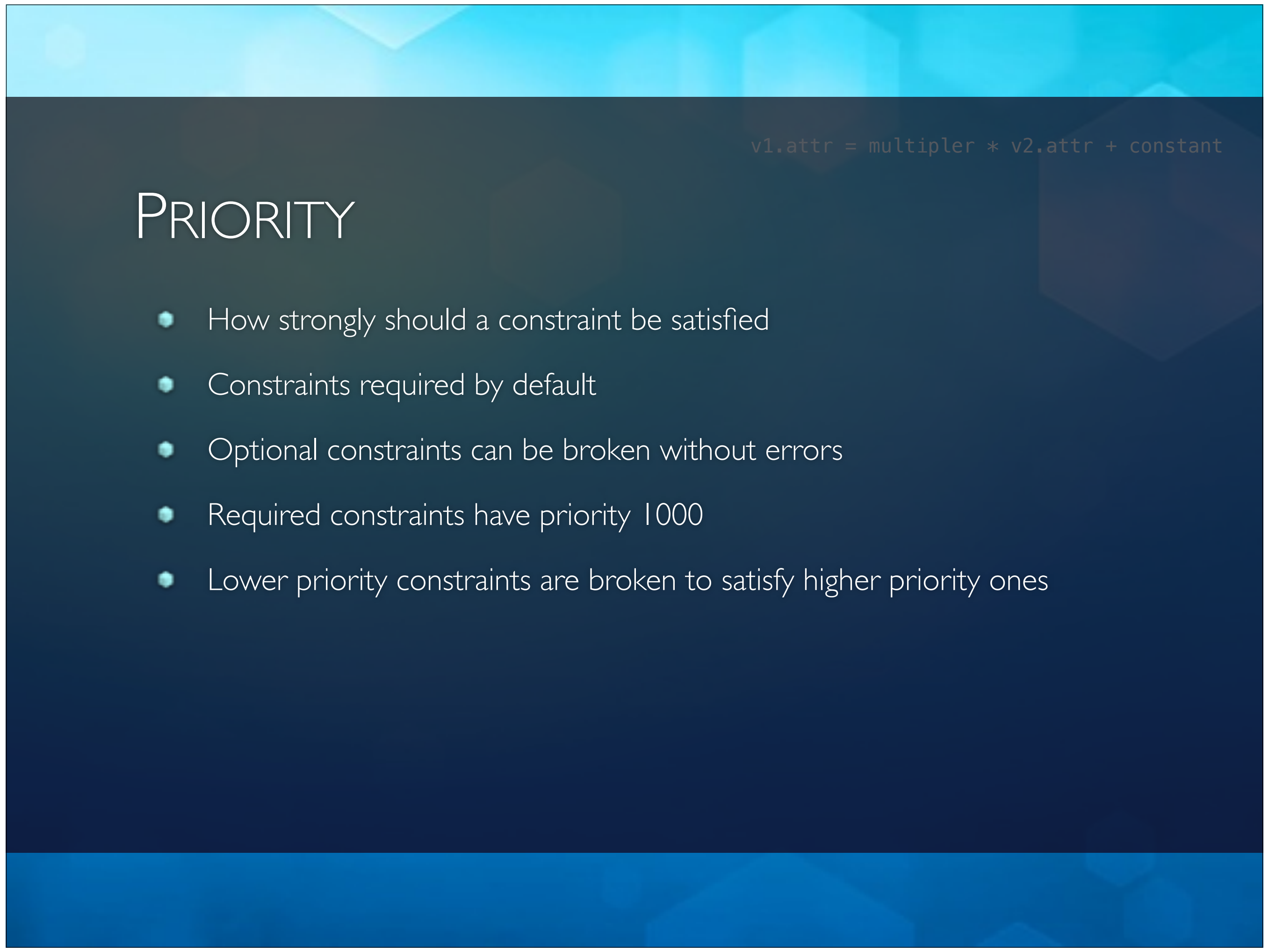
RELATIONSHIPS

- Equal
- Greater than or equal to
- Less than or equal to

`v1.attr = multipler * v2.attr + constant`

Multiplier and Constant

- Multiplier - The ratio between two attributes
- Constant - The difference between two attributes



```
v1.attr = multipler * v2.attr + constant
```

PRIORITY

- How strongly should a constraint be satisfied
- Constraints required by default
- Optional constraints can be broken without errors
- Required constraints have priority 1000
- Lower priority constraints are broken to satisfy higher priority ones

Your New Mental Model

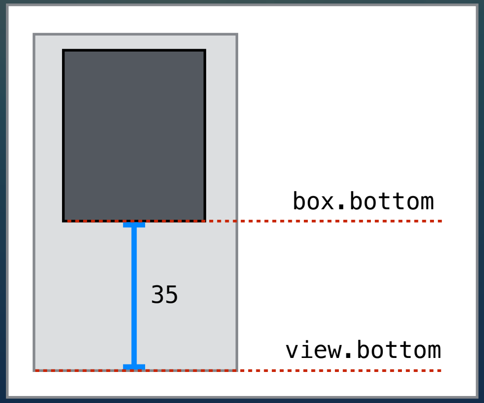
Relative vs Absolute

- Don't think in frames, think in relationships
- Most constraints are relative to other attributes
- No need to do complex calculations based on other views

Thinking In Values

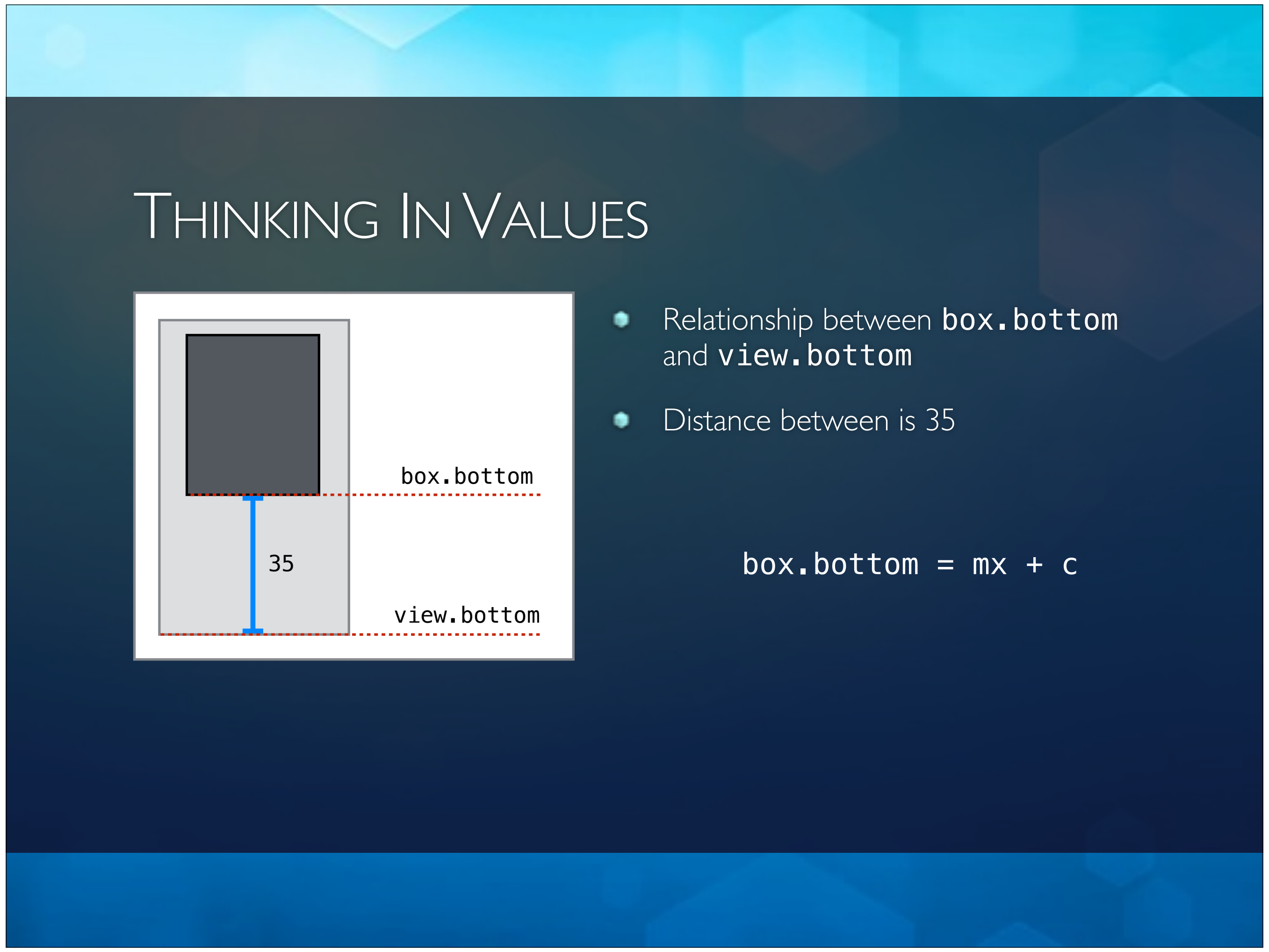
- Can be hard to work out what attributes, constant etc to use
- Don't think of them as abstract values
- Substitute in numbers

THINKING IN VALUES

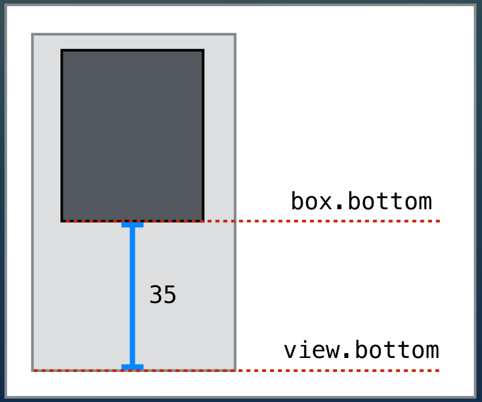


- Relationship between `box.bottom` and `view.bottom`
- Distance between is 35

```
y = mx + c
```



Thinking In Values

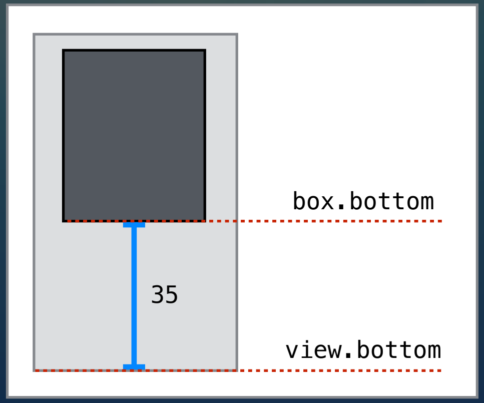


- Relationship between `box.bottom` and `view.bottom`
- Distance between is 35

```
box.bottom = mx + c
```

We want to set box.bottom so we’ll use that as “y”

THINKING IN VALUES

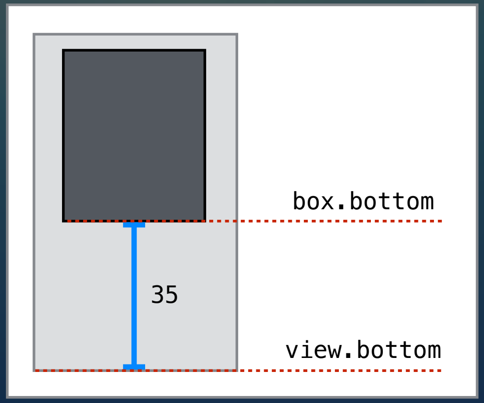


- Relationship between `box.bottom` and `view.bottom`
- Distance between is 35

```
box.bottom = x + c
```

We’re not using a ratio so m is 1, therefore we can remove it

Thinking In Values

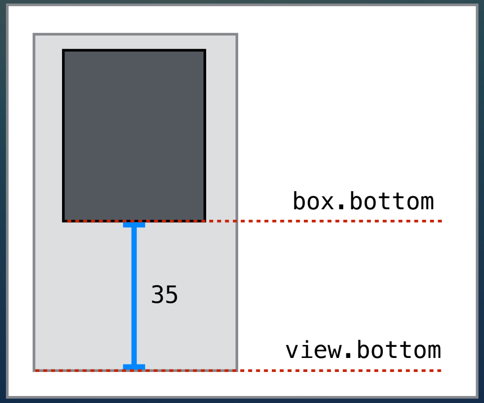


- Relationship between `box.bottom` and `view.bottom`
- Distance between is 35

```
box.bottom = view.bottom + c
```

x becomes our other attribute

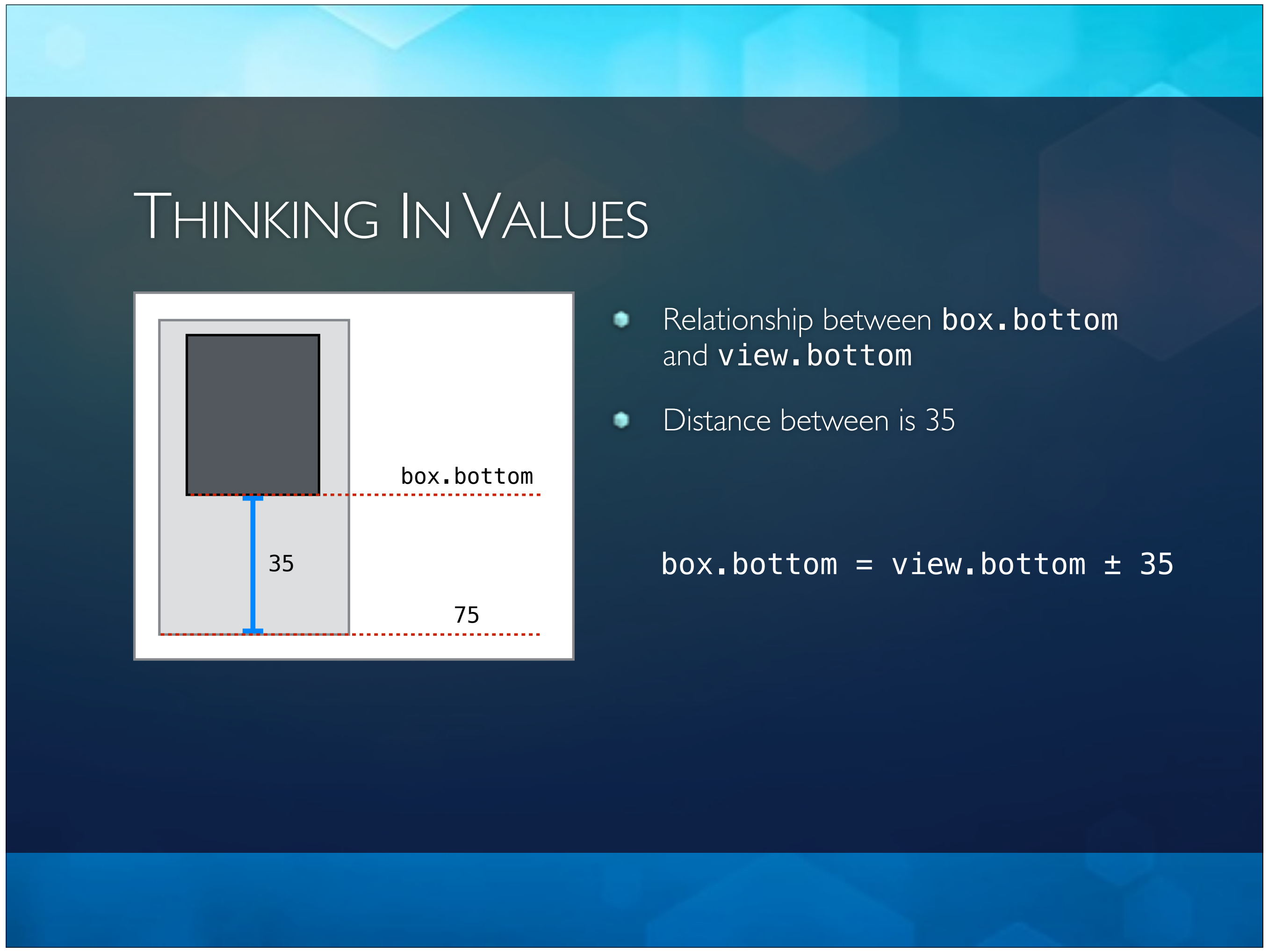
Thinking In Values



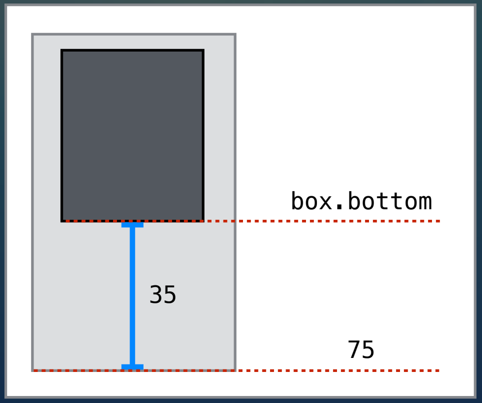
- Relationship between **`box.bottom`** and **`view.bottom`**
- Distance between is 35

```
box.bottom = view.bottom ± 35
```

We don’t yet know if c will be positive or negative 35



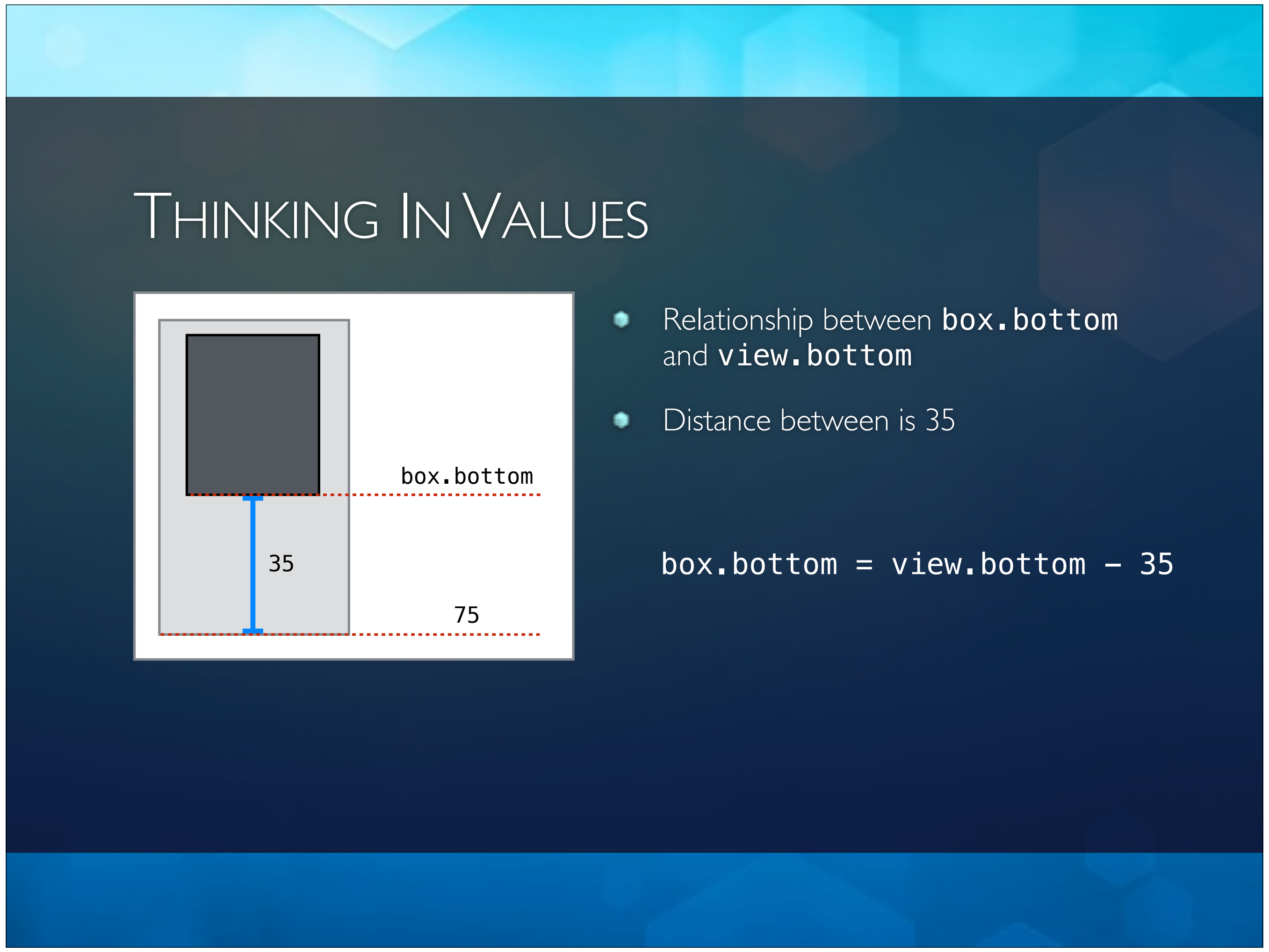
THINKING IN VALUES



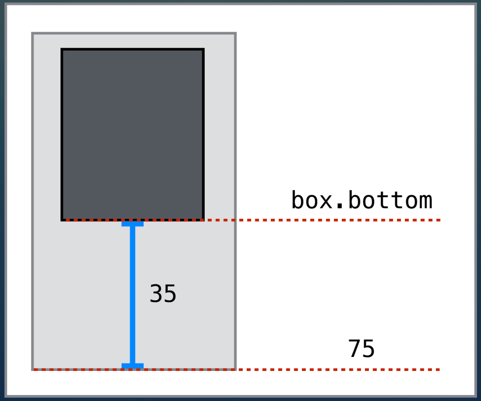
- Relationship between `box.bottom` and `view.bottom`
- Distance between is 35

```
box.bottom = view.bottom ± 35
```

We can substitute in numbers. In this case we’ll say view.bottom is equal to 75



THINKING IN VALUES

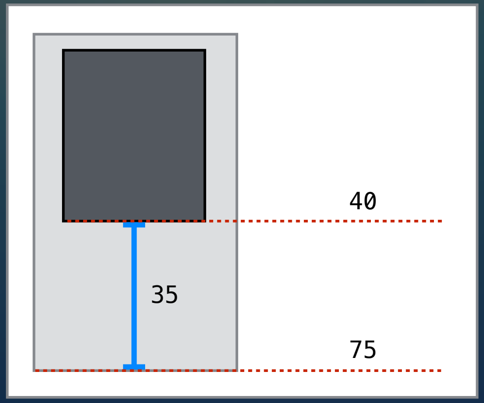


- Relationship between `box.bottom` and `view.bottom`
- Distance between is 35

```
box.bottom = view.bottom - 35
```

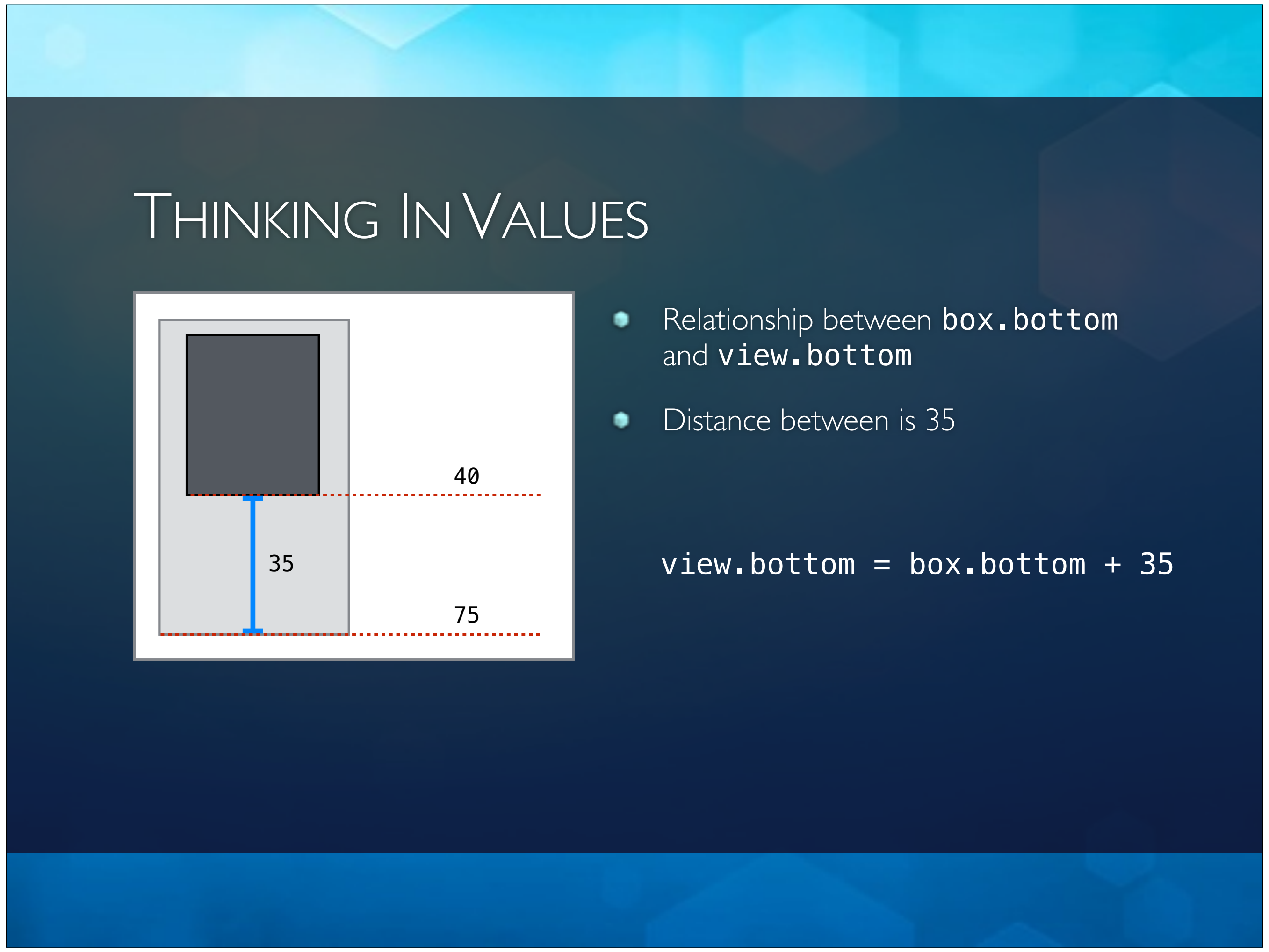
As the origin is in the top left, the y values decrease as we move up the view. As such we need to reduce the value of 75 by 35 to get box.bottom, therefore c is minus 35.

Thinking In Values

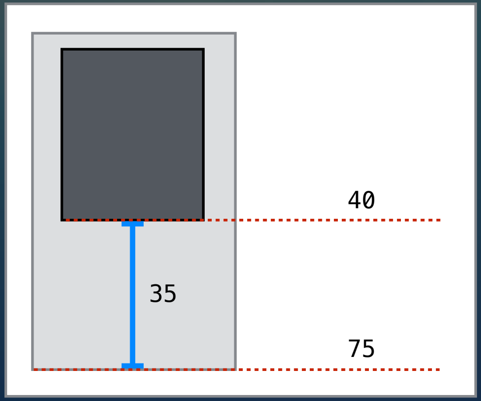


- Relationship between **`box.bottom`** and **`view.bottom`**
- Distance between is 35

```
box.bottom = view.bottom - 35
```



Thinking In Values



- Relationship between `box.bottom` and `view.bottom`
- Distance between is 35

```
view.bottom = box.bottom + 35
```

If you don’t like negative constants you can re-arrange the equation

Constraining a View

- All views need at least 4 constraints
- Need to position and size in both horizontal and vertical axes

```
leading

top

width

height
```

Constraining a View

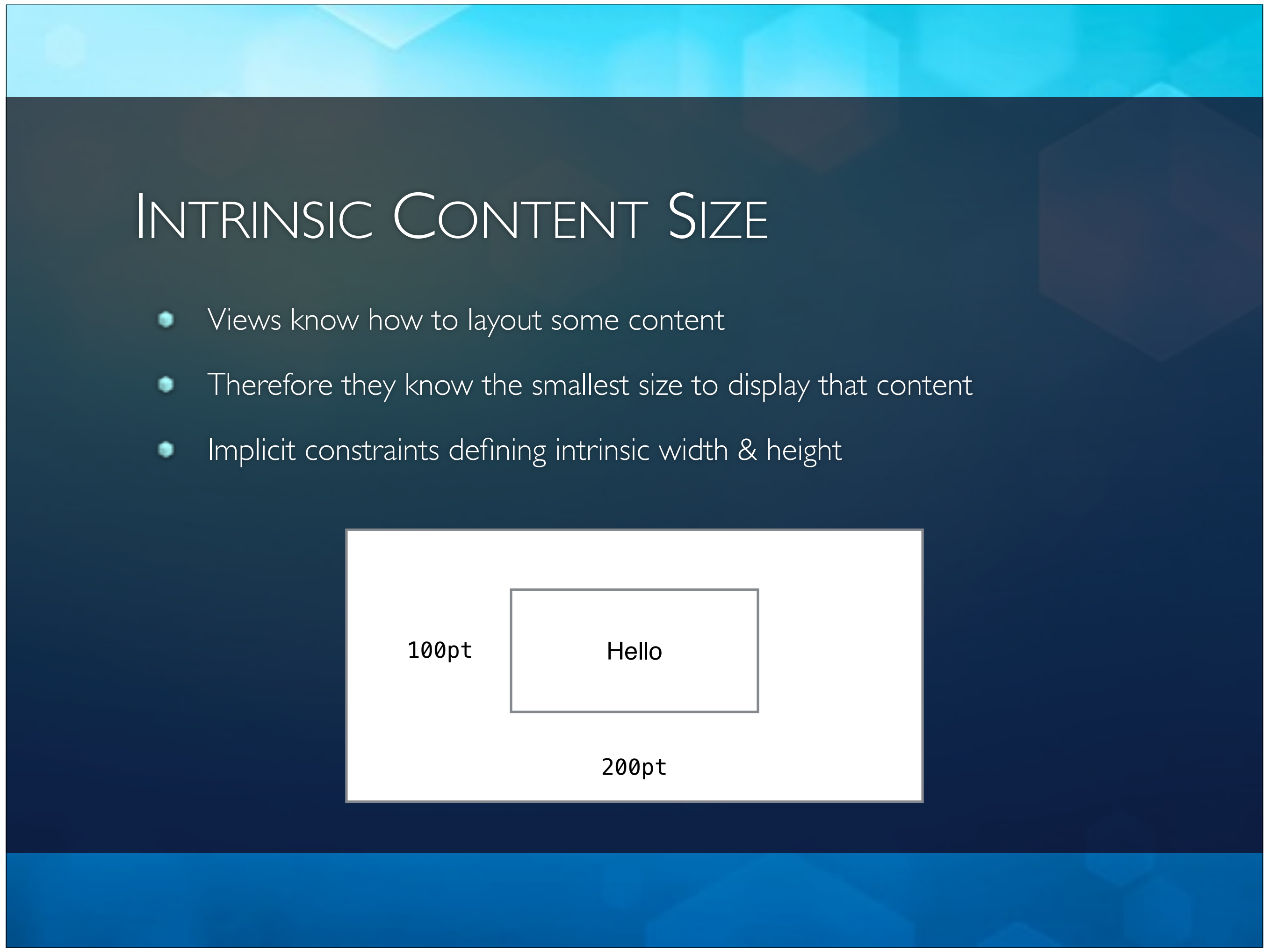
- All views need at least 4 constraints
- Need to position and size in both horizontal and vertical axes

```
trailing

bottom

width

height
```

Constraining a View

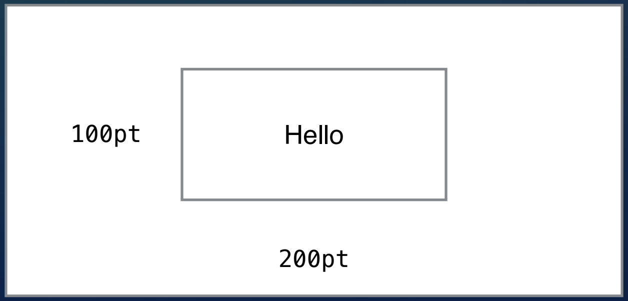
- All views need at least 4 constraints
- Need to position and size in both horizontal and vertical axes

```
top
bottom
leading
trailing
```



Intrinsic Content Size

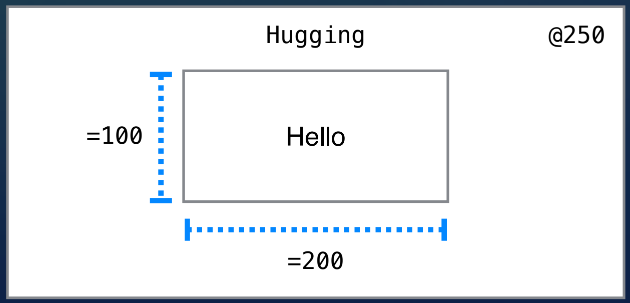
- Views know how to layout some content
- Therefore they know the smallest size to display that content
- Implicit constraints defining intrinsic width & height



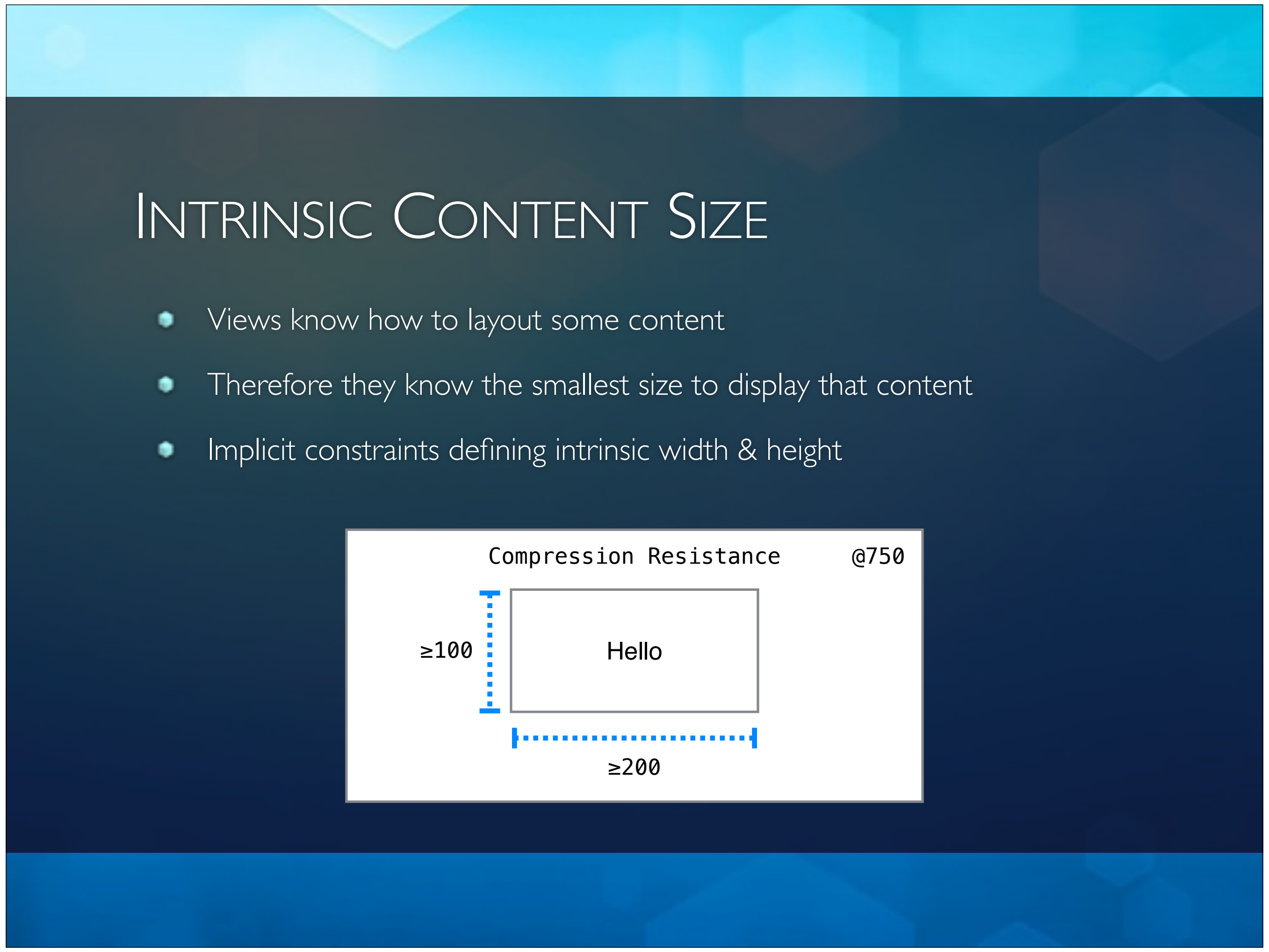
The view has an intrinsic size of 200 by 100 points based on its content

Intrinsic Content Size

- Views know how to layout some content
- Therefore they know the smallest size to display that content
- Implicit constraints defining intrinsic width & height

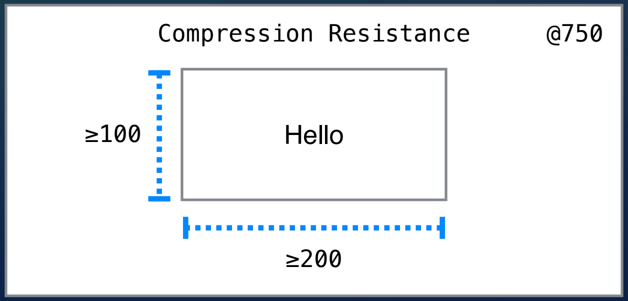


Autolayout adds implicit content hugging constraints that say the width should be equal to the intrinsic width and the height equal to the intrinsic height, so the view tries to be the smallest size possible to display its content. These are usually set at a low priority of 250



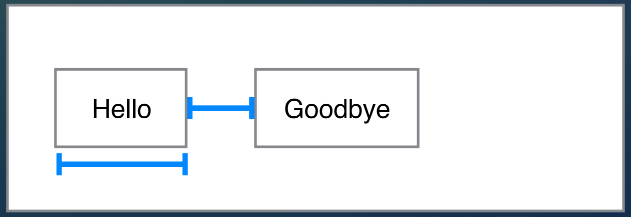
Intrinsic Content Size

- Views know how to layout some content
- Therefore they know the smallest size to display that content
- Implicit constraints defining intrinsic width & height



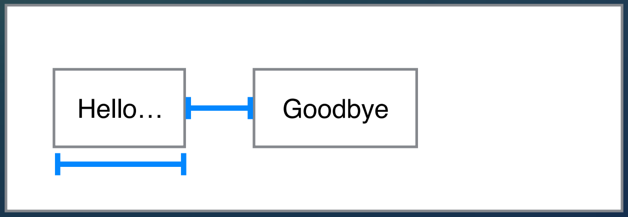
At the higher priority of 750 it adds two more implicit constraints, the content compression resistance constraints. These say the width must be greater than or equal to the intrinsic width and the height greater than or equal to the intrinsic height, so that the view does not clip its content.

Intrinsic Content Size



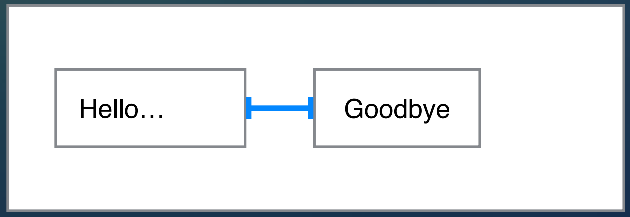
There are two buttons side by side, Hello and Goodbye. There is a fixed distance constraint between them and a fixed width constraint on the hello button

Intrinsic Content Size



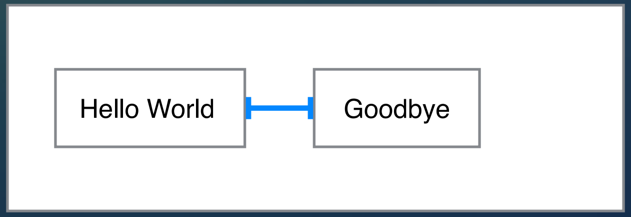
If we change the title of the hello button it will clip its content. The width constraint conflicts with the intrinsic width constraints, but due to them having a lower priority they are broken in order to satisfy the higher priority width constraint

INTRINSIC CONTENT SIZE

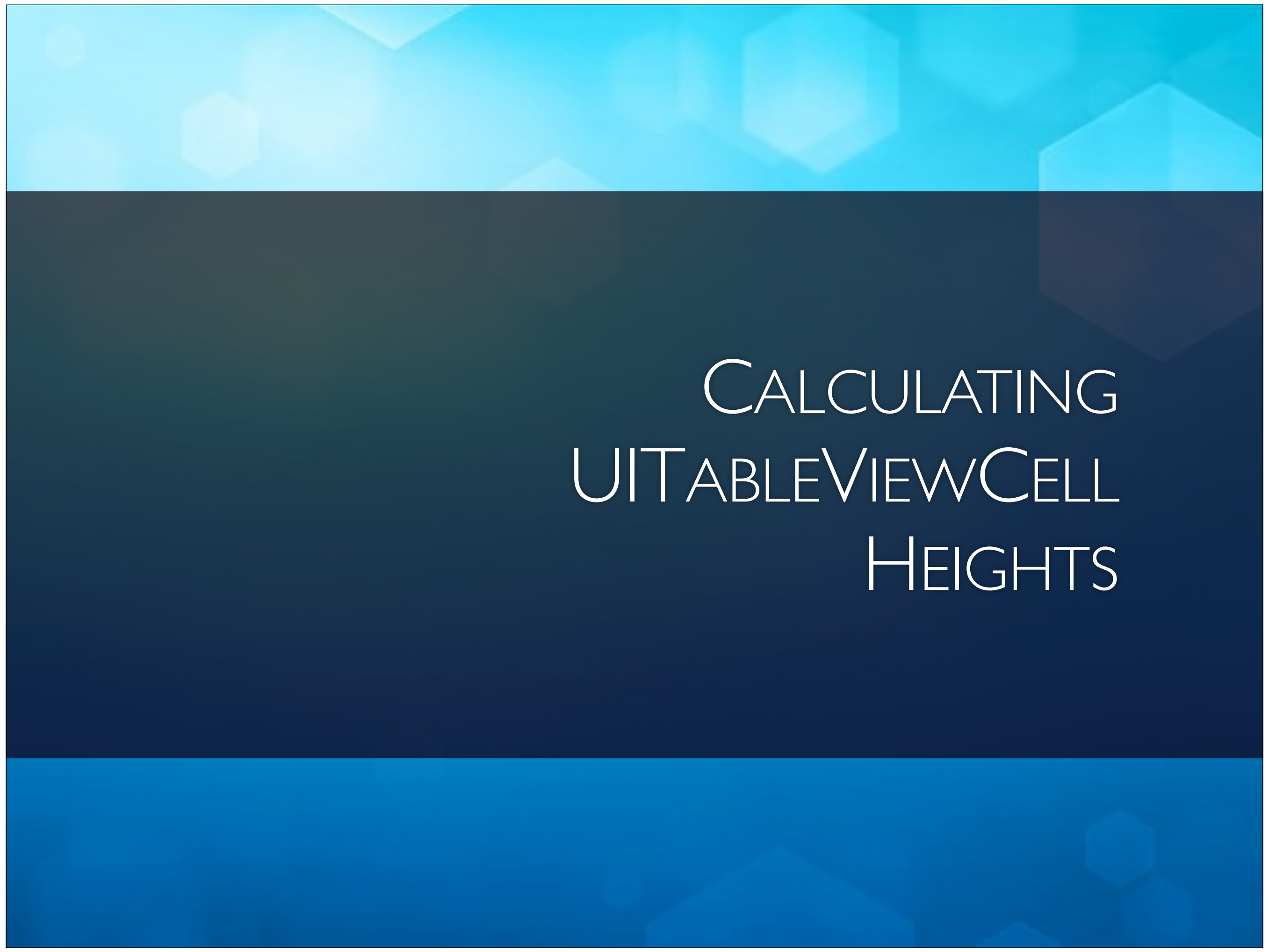


You should avoid using explicit width and height constraints as this allows views to resize to fit their content (this is what allows content-aware layout and easier localisation).

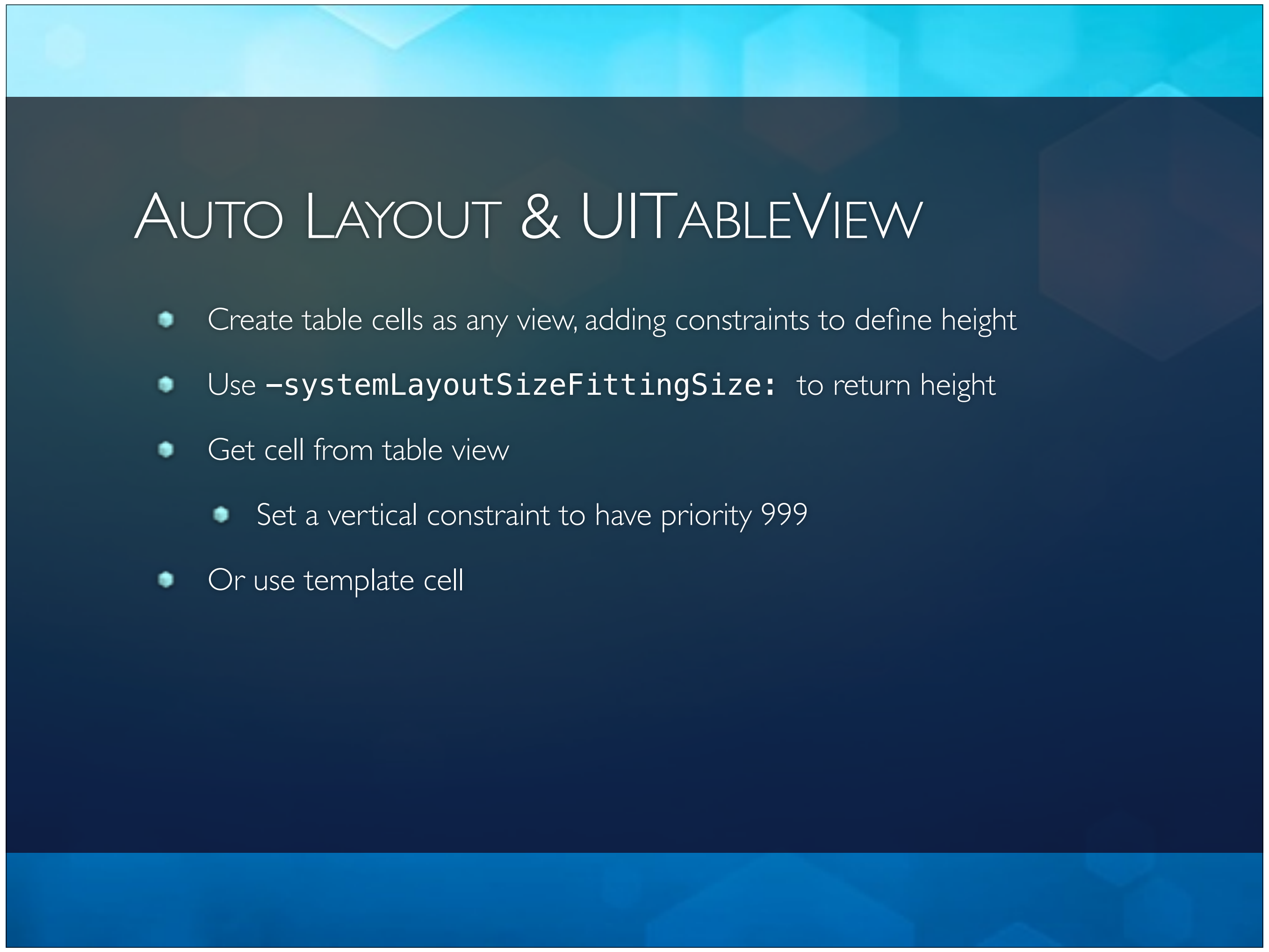
Intrinsic Content Size



As we still have the fixed distance constraint, the goodbye button will move along to maintain that distance, even though the hello button has changed its width.



Calculating UITableViewCell Heights



Auto Layout & UITableView

- Create table cells as any view, adding constraints to define height
- Use `-systemLayoutSizeFittingSize:` to return height
- Get cell from table view
  - Set a vertical constraint to have priority 999
- Or use template cell

sysLayoutSize… takes a size you want a view to be and returns the closest size it can be while satisfying constraints on the view. E.g if you want the smallest size a view can be you could pass in CGSizeZero.

One way is to get the cell to calculate it. This creates the cell and adds it though with the wrong height.
Another way is to have a template cell so you can get a size independently

Auto Layout & UITableView

iOS 8

- Create table cells as any view, adding constraints to define height
- Set `estimatedRowHeight` to most common height
- Ensure `rowHeight` is `UITableViewAutomaticDimension`

Auto-resizing UIImageView

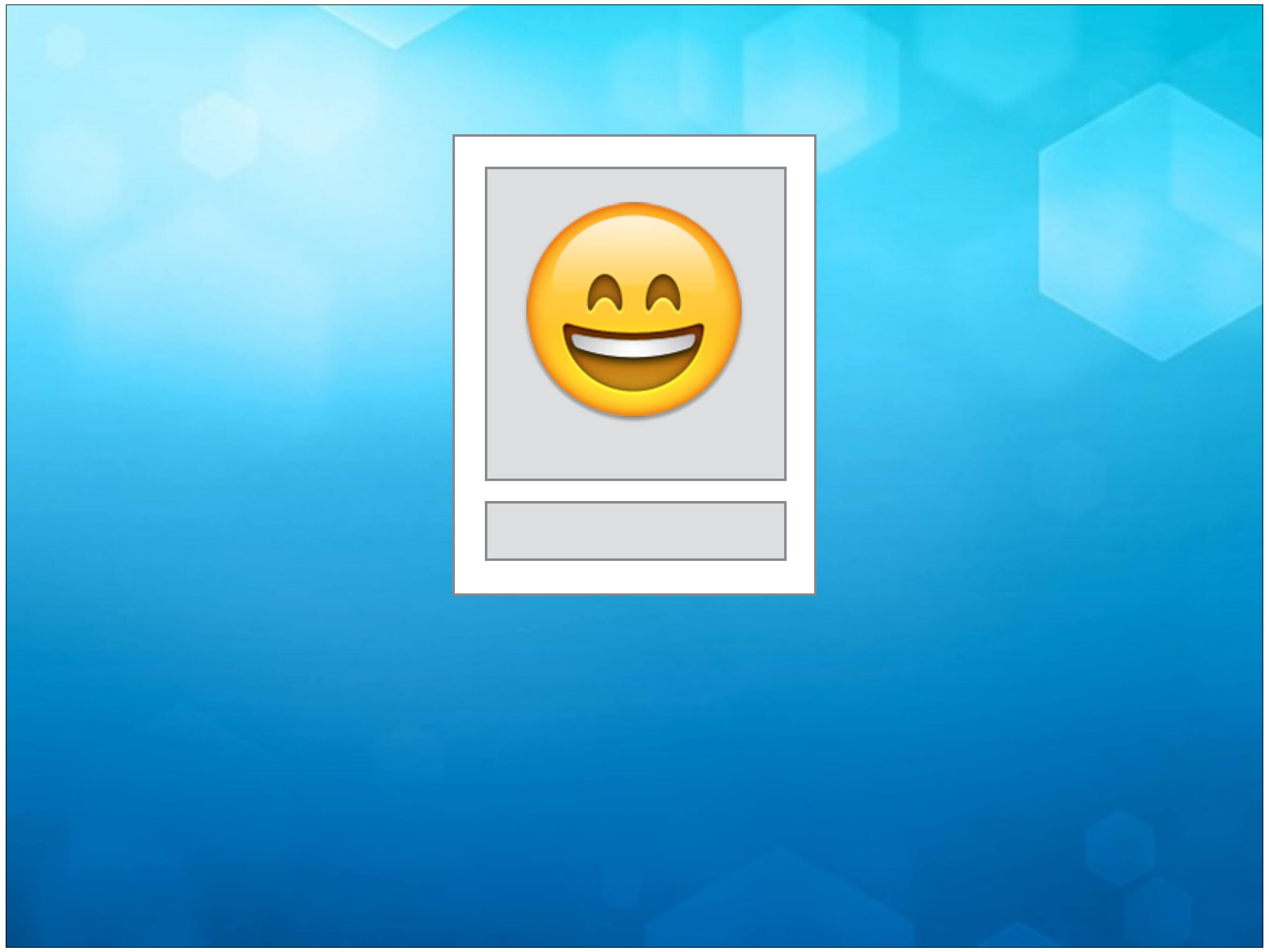
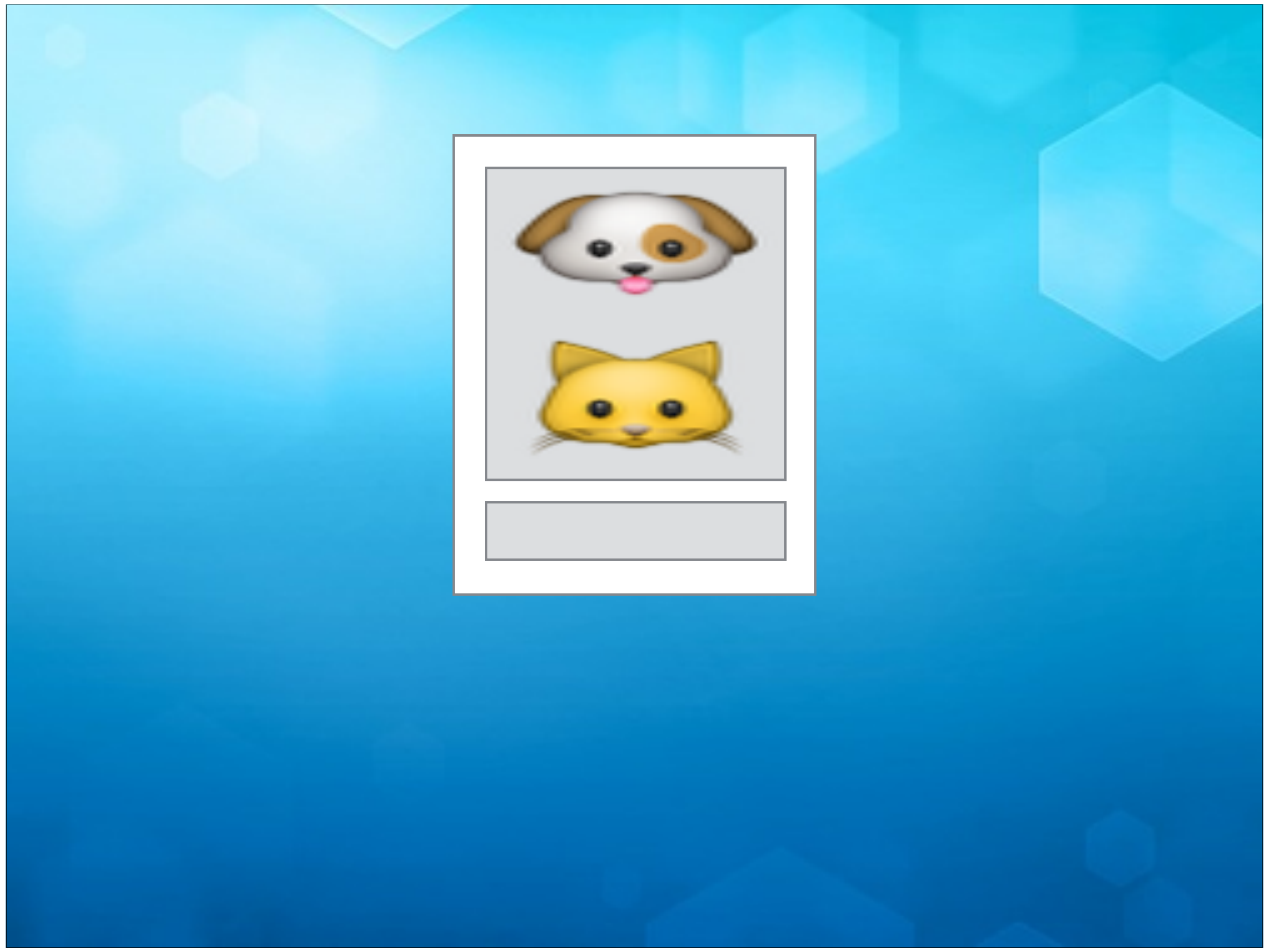


Image views don’t really handle resizing images very well.

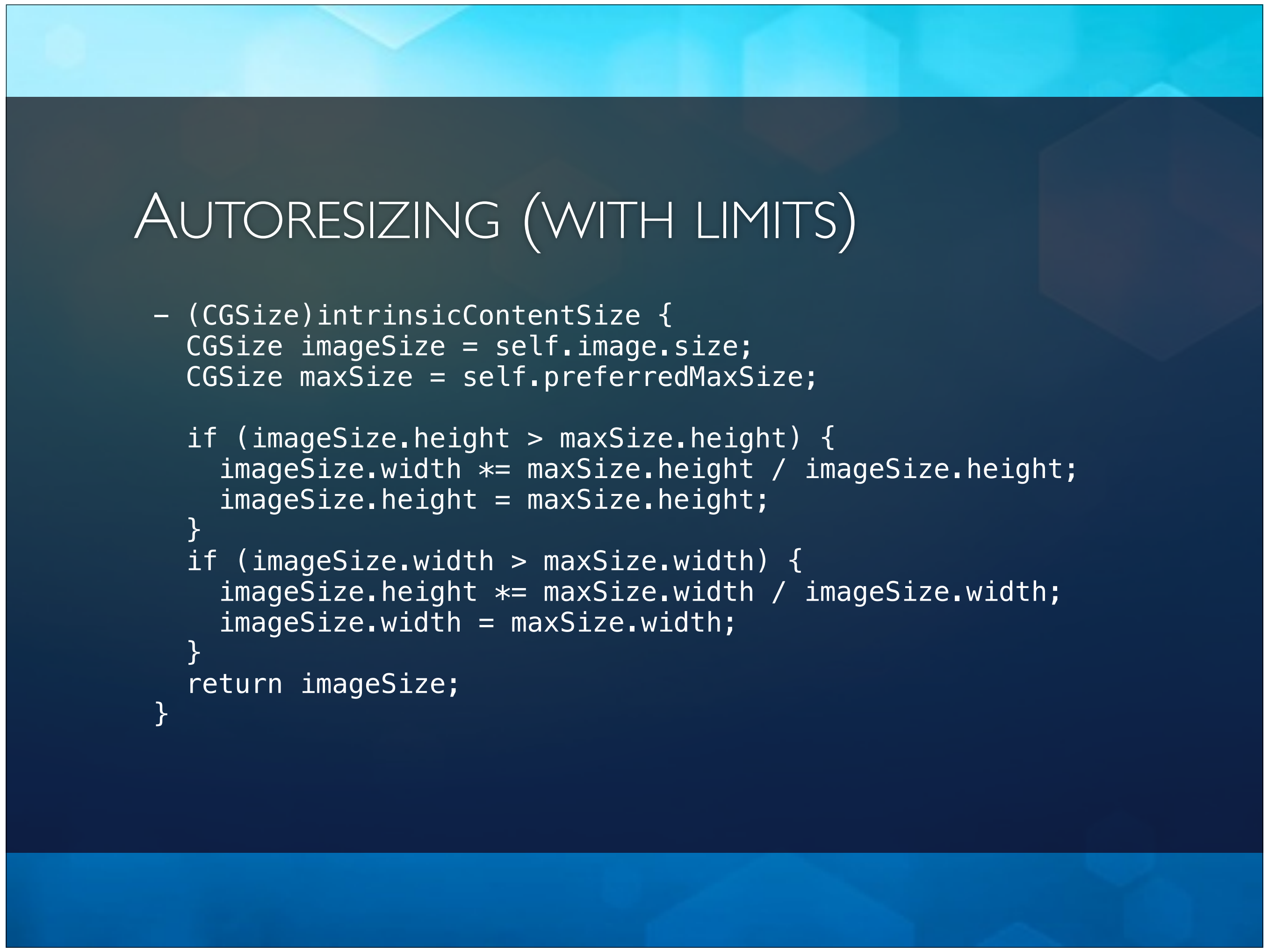


If you add a larger image it may well get squashed or stretched out of proportion

Autoresizing

- Subclass UIImageView
- Add following:

```
- (CGSize)intrinsicContentSize {
  return self.image.size;
}


- (void)setImage:(UIImage *)aImage {
  [super setImage:aImage];
  [self invalidateIntrinsicContentSize];
}
```

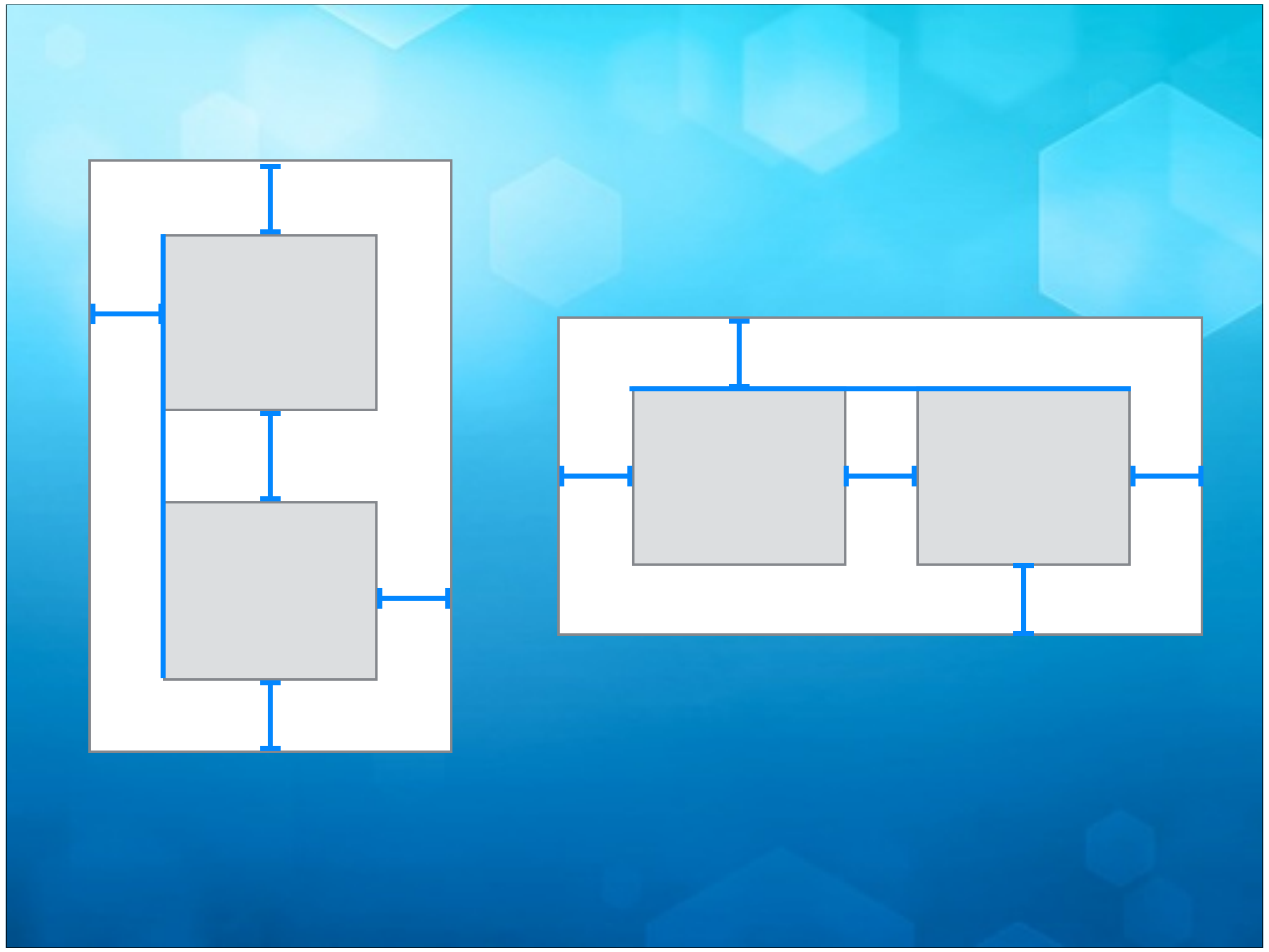


AUTORESIZING (WITH LIMITS)

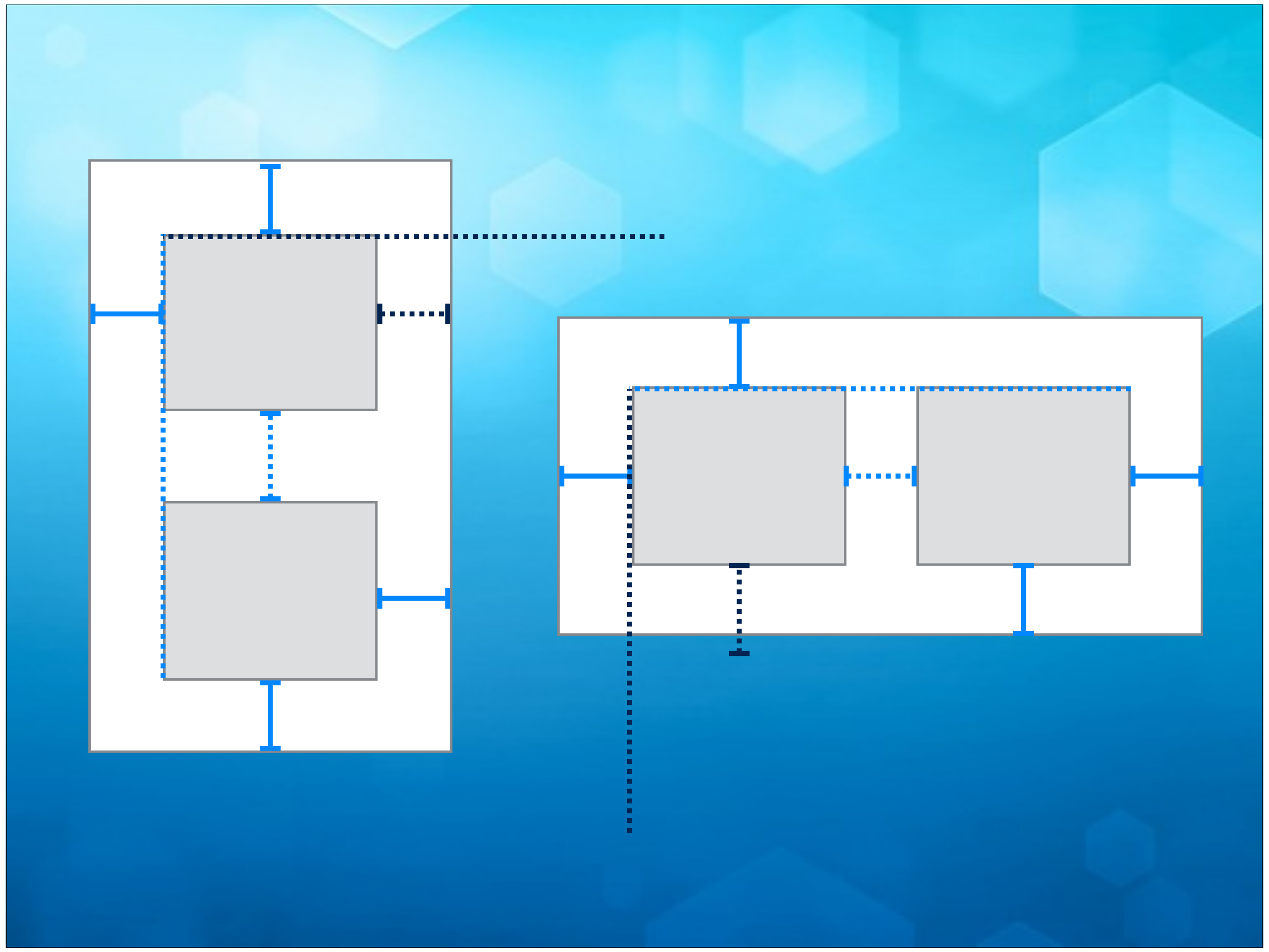
```
- (CGSize)intrinsicContentSize {
  CGSize imageSize = self.image.size;
  CGSize maxSize = self.preferredMaxSize;

  if (imageSize.height > maxSize.height) {
    imageSize.width *= maxSize.height / imageSize.height;
    imageSize.height = maxSize.height;
  }
  if (imageSize.width > maxSize.width) {
    imageSize.height *= maxSize.width / imageSize.width;
    imageSize.width = maxSize.width;
  }
  return imageSize;
}
```

This is a slightly more advanced implementation that allows us to set a maximum size.

Switching Orientation



The alignment constraints and spacing constraints between the two views conflict if they’re always required



By making them optional and changing priorities we can ensure the constraints can stay on the view but are not satisfied

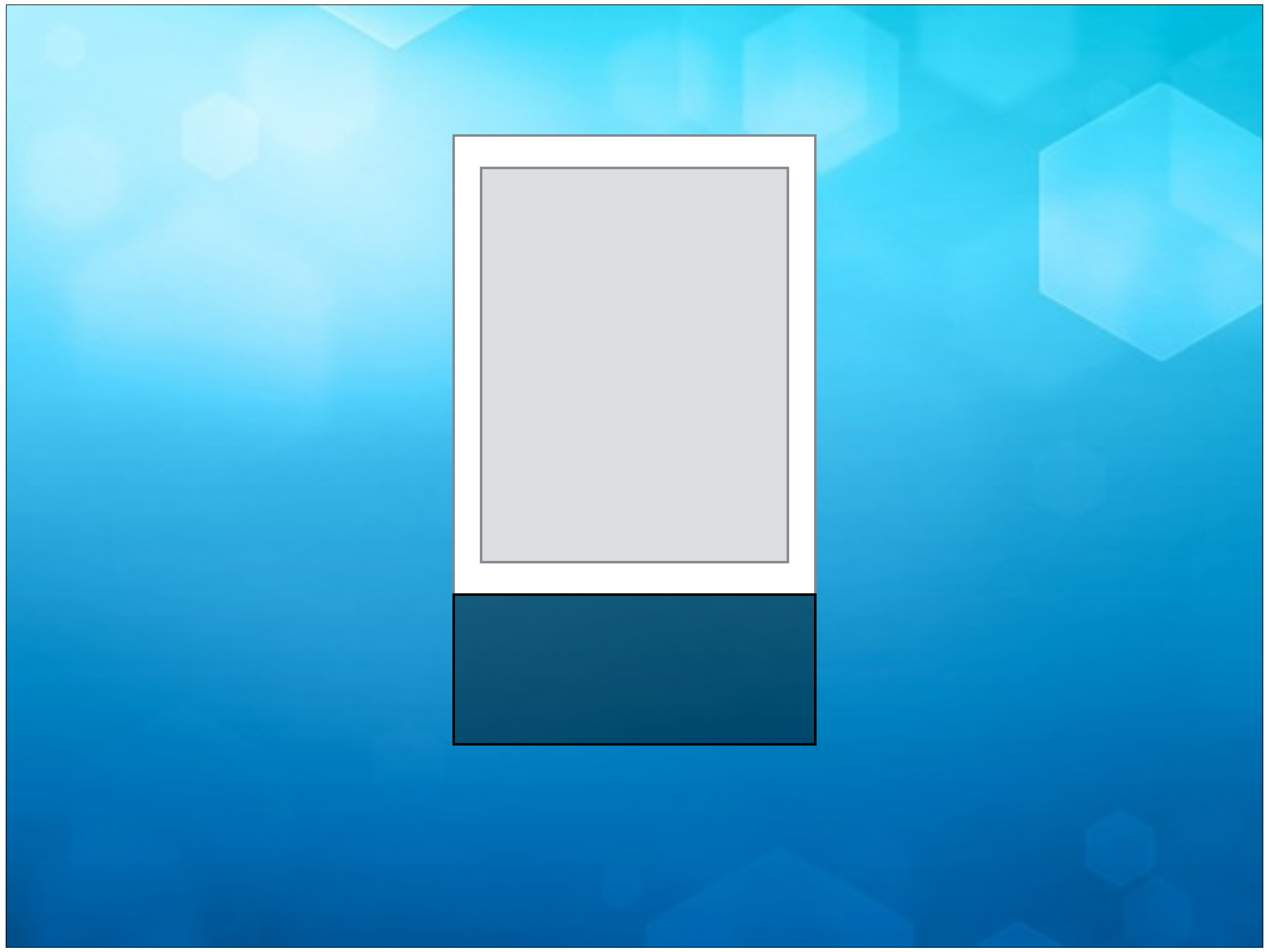
Disabling/Enabling Constraints

- Make constraints optional
- Set constraint priorities to 999 to enable
- Set to 1 to disable

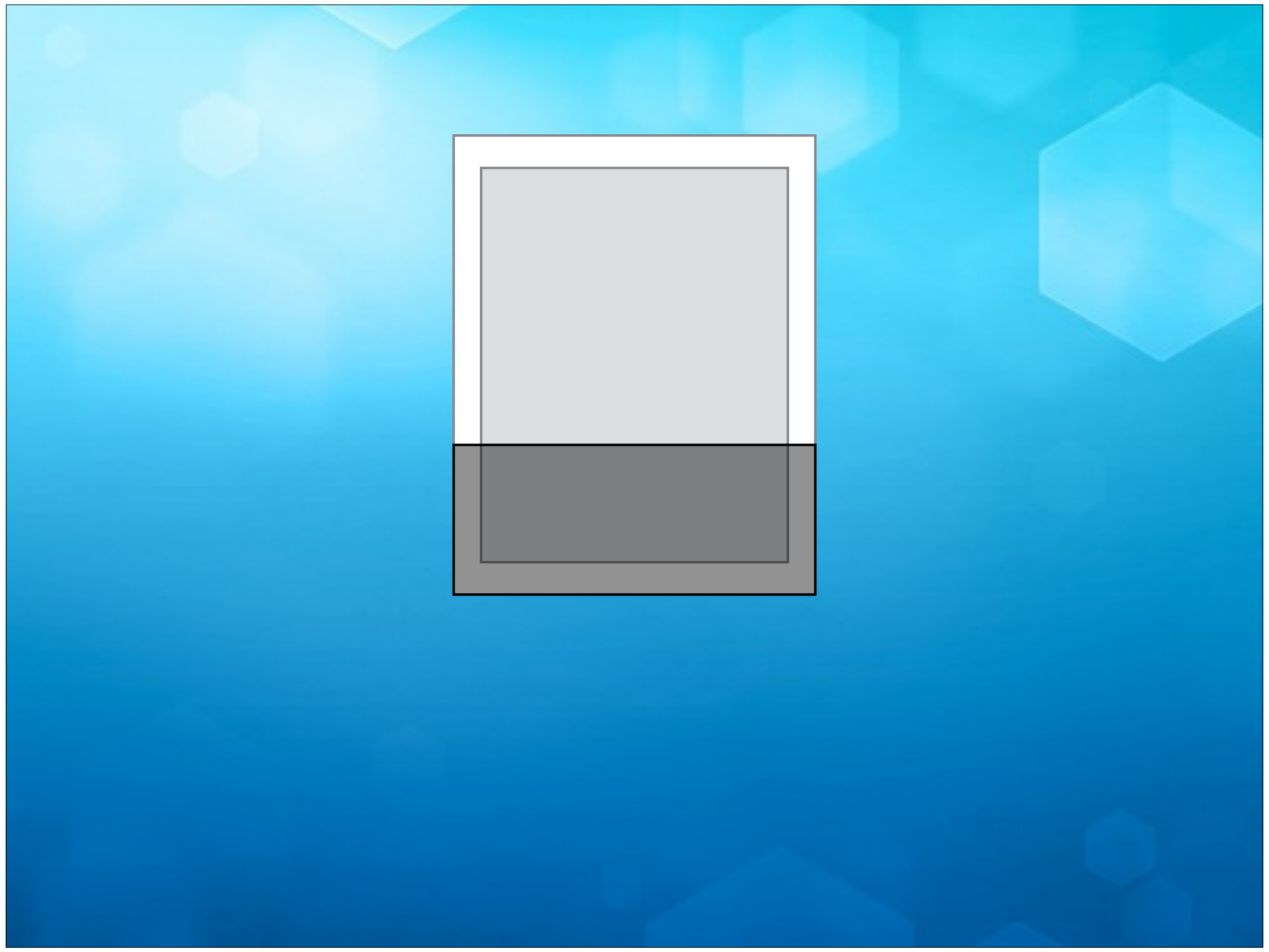
Disabling/Enabling Constraints iOS 8

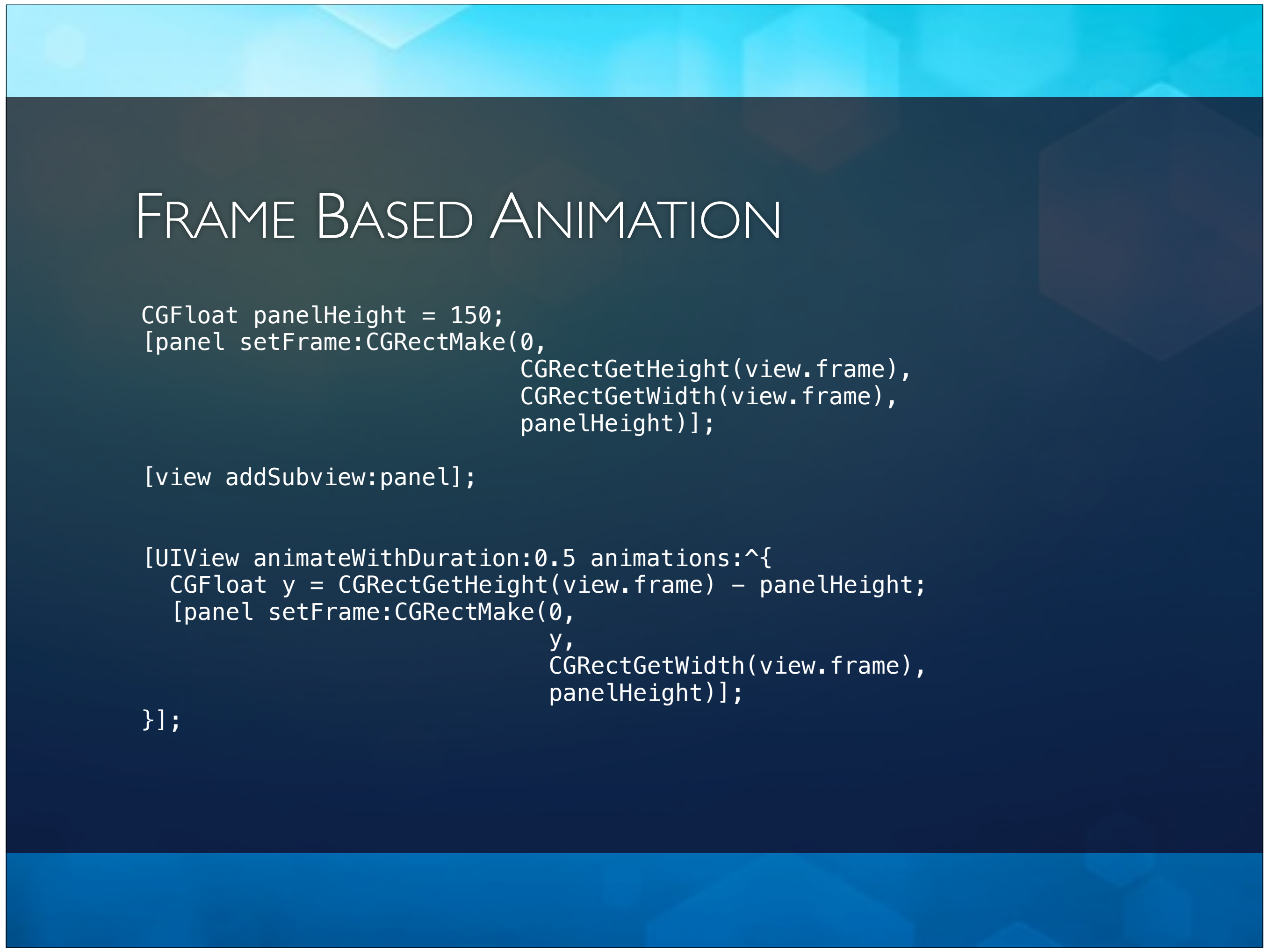
- New `active` property
- `+[NSLayoutConstraint (de)activateConstraints:]` for bulk changes
- Use NIBs with size classes

Animation



There is a view with a subview. The animation will add a panel and then slide it in over the parent and its subview.

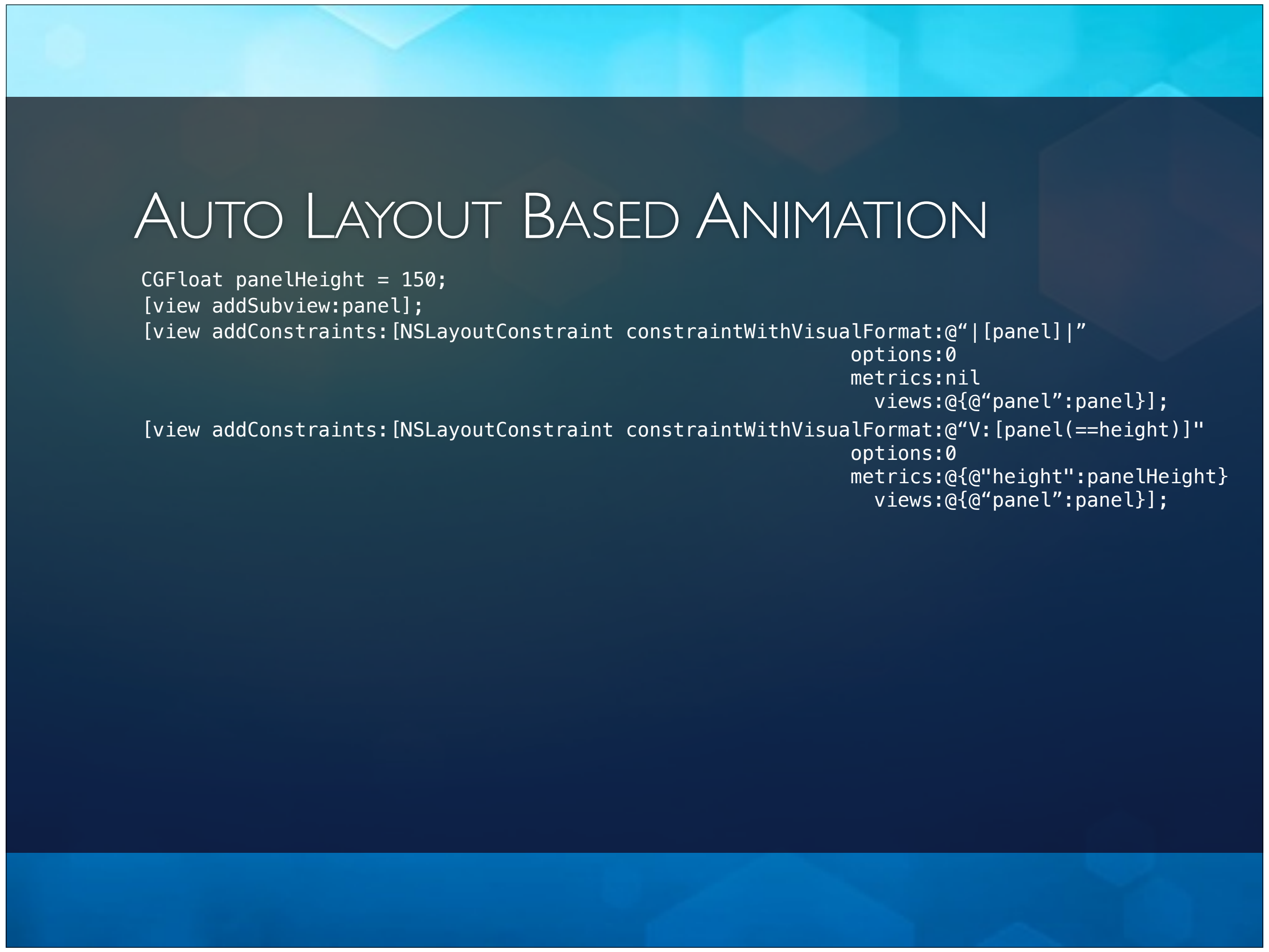




Frame Based Animation

```
CGFloat panelHeight = 150;
[panel setFrame:CGRectMake(0,
                           CGRectGetHeight(view.frame),
                           CGRectGetWidth(view.frame),
                           panelHeight)];

[view addSubview:panel];


[UIView animateWithDuration:0.5 animations:^{
  CGFloat y = CGRectGetHeight(view.frame) - panelHeight;
  [panel setFrame:CGRectMake(0,
                             y,
                             CGRectGetWidth(view.frame),
                             panelHeight)];
}];
```

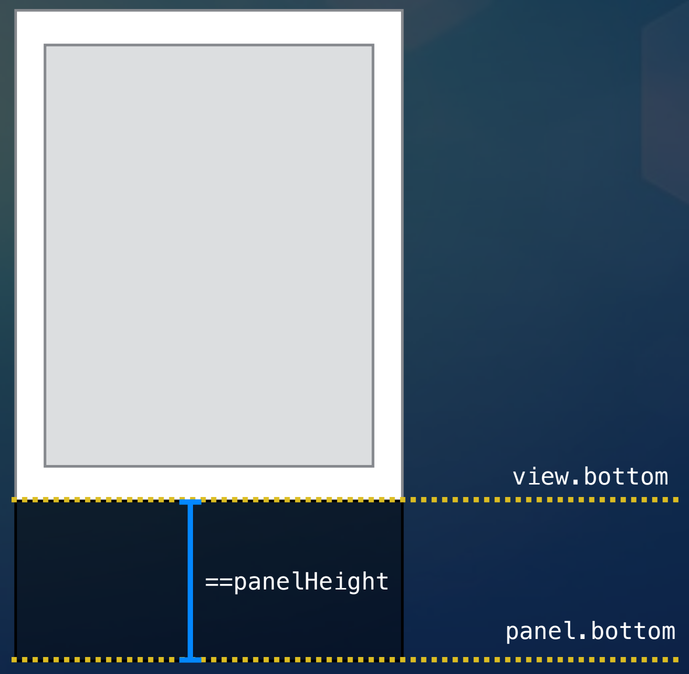
So what would a method to display this panel look like



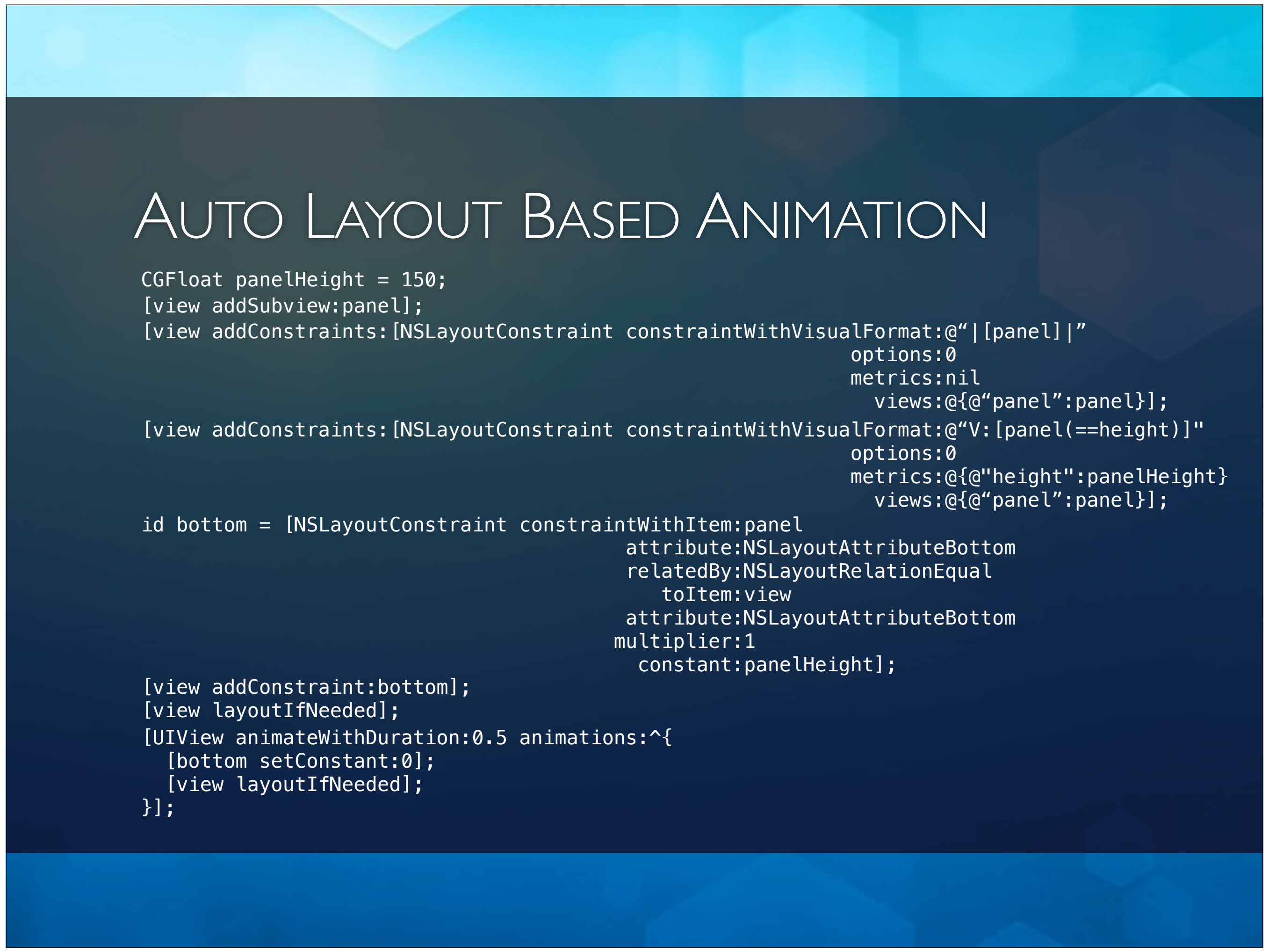
AUTO LAYOUT BASED ANIMATION

```
CGFloat panelHeight = 150;
[view addSubview:panel];
[view addConstraints:[NSLayoutConstraint constraintWithVisualFormat:@“|[panel]|”
                                                             options:0
                                                             metrics:nil
                                                               views:@{@“panel”:panel}];
[view addConstraints:[NSLayoutConstraint constraintWithVisualFormat:@“V:[panel(==height)]"
                                                             options:0
                                                             metrics:@{@"height":panelHeight}
                                                               views:@{@“panel”:panel}];
```

Lots of setup
You’ll notice we don’t care about the horizontal position or the width or the height. We only care about changing this bottom constraint (i.e. moving the panel up) which is the purpose of the animation



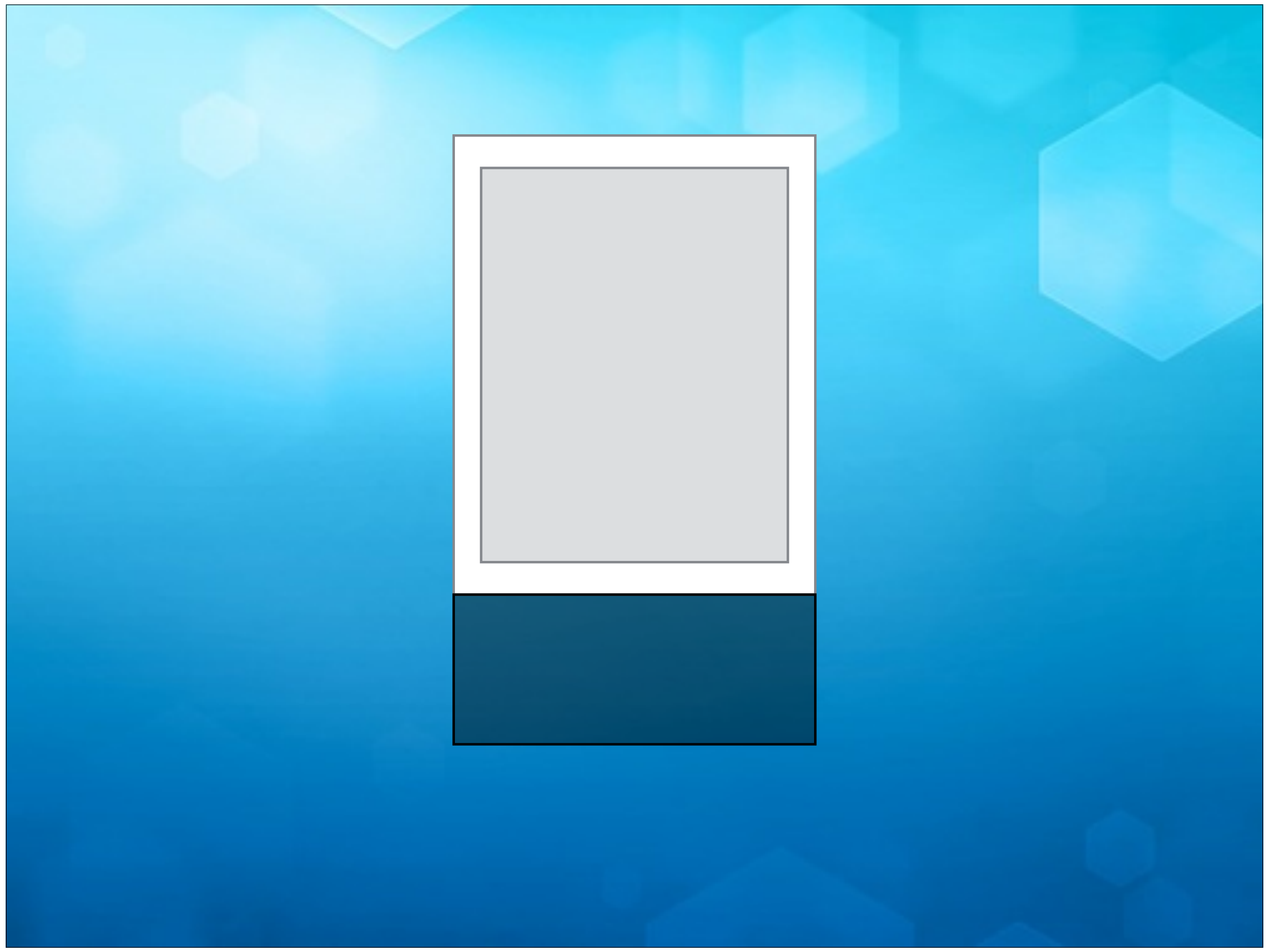
```
panel.bottom = view.bottom + panelHeight
```



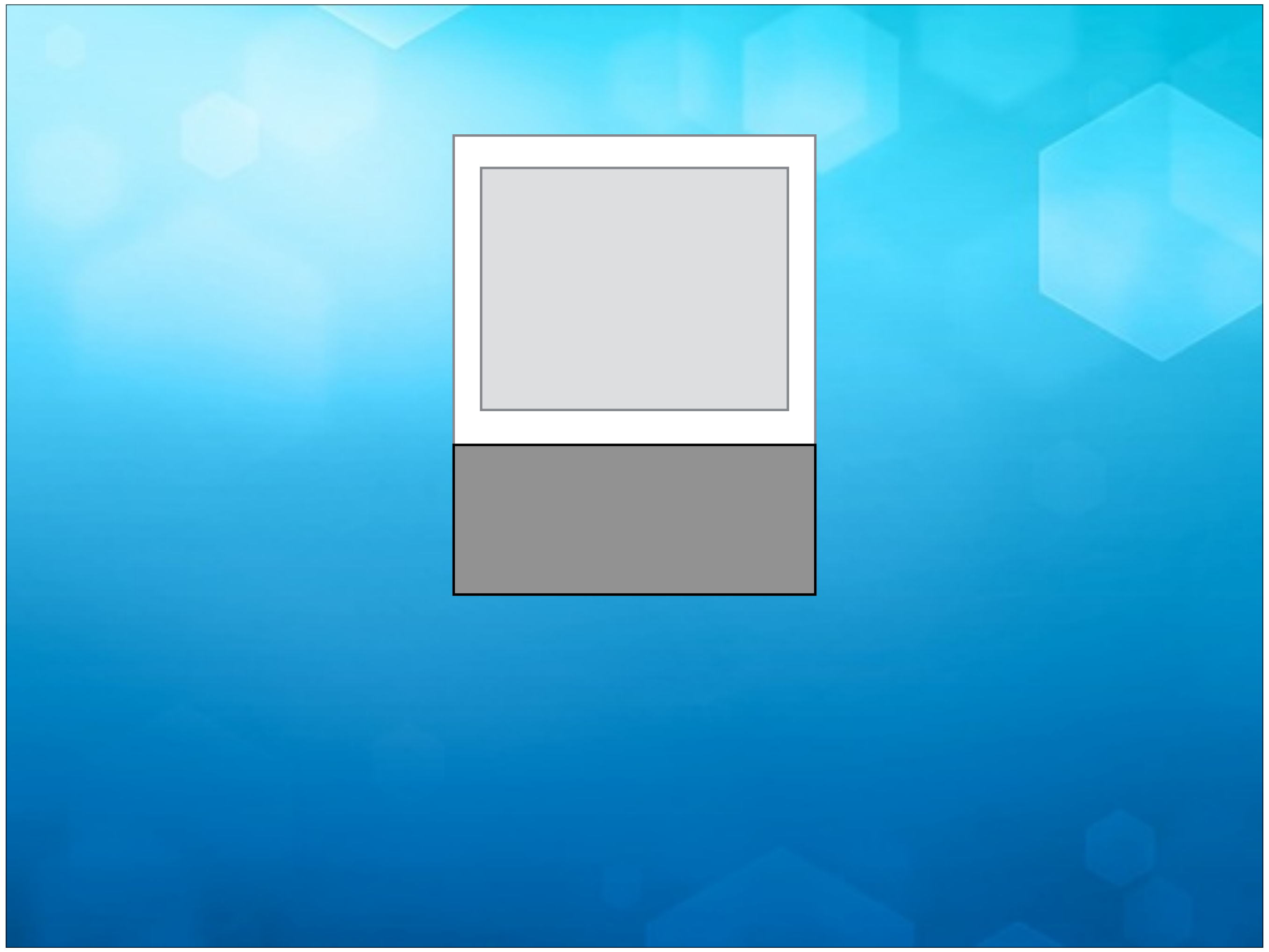
AUTO LAYOUT BASED ANIMATION

```
CGFloat panelHeight = 150;
[view addSubview:panel];
[view addConstraints:[NSLayoutConstraint constraintWithVisualFormat:@“|[panel]|”
                                                            options:0
                                                            metrics:nil
                                                              views:@{@“panel”:panel}];
[view addConstraints:[NSLayoutConstraint constraintWithVisualFormat:@“V:[panel(==height)]"
                                                            options:0
                                                            metrics:@{@"height":panelHeight}
                                                              views:@{@“panel”:panel}];
id bottom = [NSLayoutConstraint constraintWithItem:panel
                                         attribute:NSLayoutAttributeBottom
                                         relatedBy:NSLayoutRelationEqual
                                            toItem:view
                                         attribute:NSLayoutAttributeBottom
                                        multiplier:1
                                          constant:panelHeight];
[view addConstraint:bottom];
[view layoutIfNeeded];
[UIView animateWithDuration:0.5 animations:^{
  [bottom setConstant:0];
  [view layoutIfNeeded];
}];
```

Lots of setup
You’ll notice we don’t care about the horizontal position or the width or the height. We only care about changing this bottom constraint (i.e. moving the panel up) which is the purpose of the animation

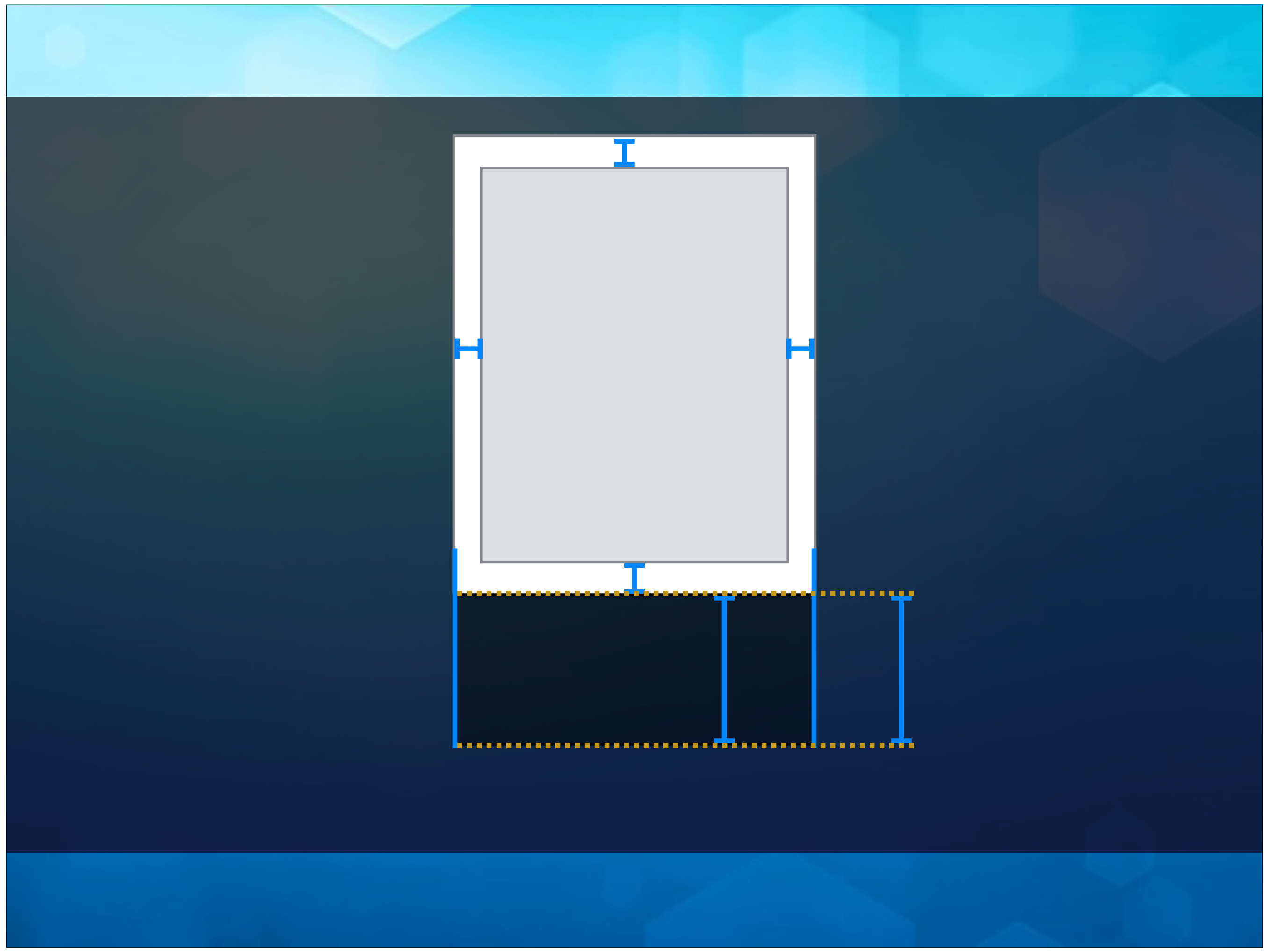


In this animation the panel is always in the view hierarchy. As it slides in, it resizes the subview (rather than appearing over it)



Frame Based Animation

```
CGFloat panelHeight = 150;
CGFloat margin = 20;

[UIView animateWithDuration:0.5 animations:^{
  CGFloat viewHeight = CGRectGetHeight(view.frame);
  CGFloat viewWidth = CGRectGetWidth(view.frame);
  CGFloat panelHeight = CGRectGetHeight(panel.frame);

  CGFloat panelY = viewHeight - panelHeight;
  [panel setFrame:CGRectMake(0, panelY, viewWidth, panelHeight)];

  CGFloat subviewWidth = viewWidth - (margin * 2)
  CGFloat subviewHeight = viewHeight - panelHeight - (margin * 2);
  [subview setFrame:CGRectMake(margin, margin, subviewWidth, subviewHeight)];
}];
```



The subview has its leading, trailing and top edges constrained to its parent. Its bottom is constrained to the panel. The panel has its leading and trailing edges constraints to its parent. Its bottom is tied to its parents bottom, as before and its height is fixed.

Auto Layout Based Animation

```
[UIView animateWithDuration:0.5 animations:^{
  [bottomConstraint setConstant:0];
  [view layoutIfNeeded];
}];
```

```
[UIView animateWithDuration:0.5 animations:^{
  [bottomConstraint setConstant:CGRectGetHeight(panel.frame)];
  [view layoutIfNeeded];
}];
```

Where To Find Me

- I code (mcubedsw.com)
- I blog (pilky.me)
- I tweet (@pilky)
- I'm writing a book (autolayoutguide.com)

QUESTIONS?