# Custom `UIControl`

Your project creates a new custom control that allows users to rate items by swiping a finger along a row of stars. This project helps you practice the concepts learned in Lambda's iOS module 6.2.

After completing the lesson material and ht is project, you'll be able to subclass UIControl to develop custom interactions.

Preview your project: https://youtu.be/kWtJLhX8-gw

## Set up the Project

Follow these steps to set up your project skeleton:

1. Create a new single-view project.
2. Drag a navigation controller into the `Main.storyboard`.
3. Delete the default table view controller.
4. Make your view controller the root view for the navigation controller.
5. Make your navigation controller the entry point for the project by setting it as the initial view controller.

## Create a new custom control class

These steps walk you through creating the new control class and adding an instance in Interface Builder:

1. Create a new Swift file using File > New File. Name it CustomControl.swift.
2. In the file, import UIKit and create a new type (called `CustomControl`) that you subclass from `UIControl`.
3. Add a new Int-typed variable property called `value` to your class. It's initial value should be 1. This property is API-facing, so clients will be able to see it. It establishes your control as a value-providing (and value-changing) type.
4. In Interface builder, add a new view. Change the background color in the Attributes Inspector so it's visible. Don't worry about the color you pick. This is just to make it easier to work with.
5. Use the Identity Inspector to set the class to Custom Control.
6. Center it with Auto Layout but don't set any rules about size. Instead, your type will use an intrinsic size to tell Auto Layout how big it will be.
7. In the Size Inspector, select Ambiguity > Verify Position Only. This supports your "no-size" layout.
8. Use Ctrl-drag to connect your view to ViewController.swift with an IBAction. Note the "Event" pop-up currently set to "Value Changed". Look at the other options in the pop-up (like "Touch Down" and "Touch Drag Inside") but keep the event set to "Value Changed". Name your IBAction `updateRating`. This method allows the control's client (in this case your view controller) to receive updates about changes in the rating control.
9. Edit the new method's signature to: `@IBAction func updateRating(_ ratingControl: CustomControl)`. This keeps you from having to cast the `sender` to the right class.
10. Implement `updateRating`. Set the view controller's title to the string `"User Rating: N stars"` where N is the number of stars. This number is the control's visible `value` property.
11. **Stretch**: Fix the title so it's correct for 1 ("star" not "stars") as well as 2-5.

Target-action associates callbacks with an event. IB handles that for you by calling `func addTarget(_ target: Any?, action: Selector, for controlEvents: UIControlEvents)` on your behalf. (You can add this by hand but it's so much easier to use IB.) This callback is a combination of an instance being called (the "target") and the method on the instance (the "action").

## Build the control view

Custom controls are often fixed size (although they may use different sizes on iPhone and iPad targets). You'll create a control that consists of five labels. These are built without Auto Layout using a view's `frame` property to specify where each subview lies. Since the control will never change size, it's safe to use absolute numbers for each frame.

All this work takes place in your CustomControl.swift file.

1. Create the following private constants for your class: `componentDimension`, a CGFloat equal to 40.0, `componentCount`, equal to 5, `componentActiveColor`, which is `UIColor.black`, and `componentInactiveColor`, which is `UIColor.gray`.
2. Add an `init?(coder aCoder: NSCoder)` initializer. After calling `super`, have it call a `setup()` function, which is where you'll perform your setup work.
3. In `setup` use a loop to create five labels (using the `UILabel()` constructor). Add each one as a subview. Store each label into a local array with `append`.
4. In your loop, add a tag for each view that represents which star it is. The first star is tag 1. The fifth is tag 5. The tags let you quickly update the control's value.
5. Set each label's frame to size `componentDimension` by `componentDimension`. (Yes, they are all square). Lay out the labels in a row with 8 points of space between each one. The first label should be at (8.0, 0), which allows a small pad between it and the edge of the control. The next one starts at (componentDimension + 16.0, 0.0), and so forth.
6. Set the font (bold system font, size 32.0), text (pick your favorite Unicode star from the character picker), and alignment (center) for your label.
7. Set the label's `textColor` to either the active (for the first) or inactive (for the others) component color.
8. Add the following method to your type. This method tells Auto Layout how big your custom control should be.

```
override var intrinsicContentSize: CGSize {
  let componentsWidth = CGFloat(componentCount) * componentDimension
  let componentsSpacing = CGFloat(componentCount + 1) * 8.0
  let width = componentsWidth + componentsSpacing
  return CGSize(width: width, height: componentDimension)
}
```

Test and run your application to make sure your control is visible and that the stars are laid out as expected.

## Add touch handlers.

Follow these steps to add touch to your control class. These handler implementations are almost identical to the ones in your guided project. You'll be adding:

- `beginTracking(_ touch: UITouch, with event: UIEvent?) -> Bool`
- `continueTracking(_ touch: UITouch, with event: UIEvent?) -> Bool`
- `endTracking(_ touch: UITouch?, with event: UIEvent?)`
- `cancelTracking(with event: UIEvent?)`

The `sendActions(for:)` method you'll call from these handlers is what makes a control a control. Sending actions generates the events that you use in IB with target-action. If you want to subscribe, say, to a `touchUpInside`, you'd control-drag and then choose *that* event from the IBAction `Event` popup.

Your code will demonstrate the kinds of events that can be subscribed to and are typical for UIControls. As a rule, always produce as many sent actions as possible because you don't know how your controls will be used in the future. The more exhaustive you are, the better the shelf life of your controls. Plus it's (1) minimal code and (2) almost boilerplate. You can reuse this code between controls with little change.

1. Add skeletons for `begin`, `continue`, `end`, and `cancel` tracking methods. The `begin` and `continue` methods should just return true.
2. Add a skeleton for an `updateValue(at touch: UITouch)` method.
3. In `cancel`, send an action for `.touchCancel`
4. In `begin`, add `updateValue()` to respond to the start of your user's touch.
5. In `continue`, check whether the touch is inside view bounds. Send `.touchDragInside` or `.touchDragOutside`. If the touch is inside, update your value with `updateValue()`.
6. In `end`, make sure you still have a valid touch and return if not. Otherwise, check whether the touch is inside view bounds and duplicate the logic for `continue`. Replace the two actions with `touchUpInside` and `touchUpOutside` instead of the drag forms you used for `continue`.

In your code, the `end` handler generates a value update. It provides a little safety net in case the lift event has moved the finger in an untracked movement. You can omit it if desired or keep it if you feel cautious.

With value controls, there's no penalty for spawning extra `.valueChanged` events with `sendActions(for:)`. Be far more cautious with trigger controls. A button or other trigger should only send *one* primary action. For these, keep a "wasTriggered" Boolean variable on hand. Do not `sendActions` after the first control trigger. Otherwise, your angry customers may end up authorizing multiple payments or using up all the arrows in their quiver when they only intended to pay once or shoot once.

## Respond to touches.

Follow these steps to finish up your control and add user feedback.

1. In `updateValue`, you handle touches by checking to see whether they intersect with any of your stored label subviews. Implement a loop that iterates through your component labels and detect whether each touch's location (`touch.location(in: self)`) is contained in each label's frame.
2. When a touch overlaps a label, set the control's value to that tag, update the label colors to reflect the current touch, and send an action for `valueChanged`.
3. **Stretch** It's better to store the old value before changing it and only send an update when the value has changed.
4. **Stretch** Add the following `UIView` animation to flare the view a little when selected.

```
extension UIView {
  // "Flare view" animation sequence
  func performFlare() {
    func flare()   { transform = CGAffineTransform(scaleX: 1.6, y: 1.6) }
    func unflare() { transform = .identity }

    UIView.animate(withDuration: 0.3,
                   animations: { flare() },
                   completion: { _ in UIView.animate(withDuration: 0.1) { unflare() }})
  }
}
```

This method adjusts your view's `transform`, telling it to inflate to 60% larger than its normal size and then shrink back down to its default size (using the "no change" `identity` transform).

## Test

1. Run the project and make sure everything works.
2. If anything doesn't work the way the video shows (outside of the two stretch goals), go back and debug your issues.
3. As always, if you need help, follow the 20-minute rule, then ask your PM.

## Go Farther

Time allowing, here are some things you can try.

1. Change the number of buttons to allow 4 or 6 star rating controls. Does everything work right with your change being in a single line of code?
2. Flip the control for use in right-to-left language countries, where you slide from right to left.
3. Create different flare animations. Things you can animate include background color, translucency (the view's `alpha` property), and rotation. You'll be able to find examples of these with simple web searches.

## References

Here are some helpful resources for your project:

- `UIControl` documentation (Xcode)
- `UIControlEvents` documentation (Xcode): a list of all available control events, including the "value changed" event used in this challenge.
- How to build selectors by hand (apologies for the URL): http://fuckingselectorsyntax.com
- Examples of custom controls: Search the web for `uicontrol site:github.com`
- Custom Cocoa Control repository: https://www.cocoacontrols.com