Chapter 15: Dynamic Type

By Jerry Beers

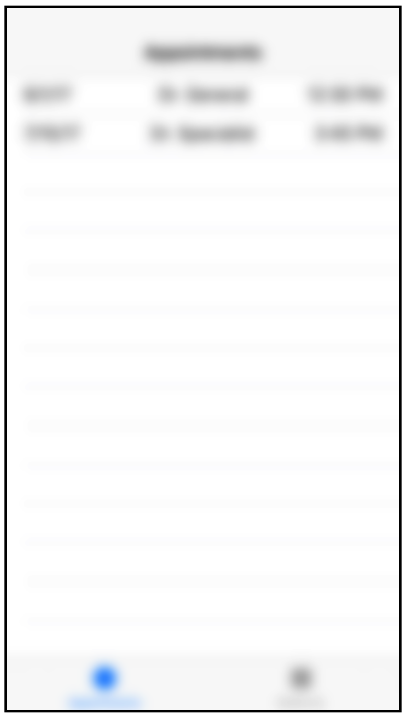
Have you ever looked at someone else's iPhone or iPad and noticed that their text size is different from what you're used to? Maybe they have the size turned up to make it easier for them to see. Or maybe they have smaller text to fit more content on the screen.

This is a really thoughtful feature from Apple to allow users to customize their device to the text size that suits them best. But if your app doesn't play along, your users won't get the benefit of this setting.

It's handy for a user to be able to specify if they want smaller text, but this feature is more important than that. The sample app for this chapter, MedJournal, is an example of the important information people have come to rely on their iPhones for.

Open the starter project, build and run, and you'll see the main interface of the app. From anywhere in the app, if you triple tap, the app will show a blurred snapshot of the interface (the blurring process may take several seconds).

> **Note**: On the Simulator, the blurring process can take as long as 60 seconds. Be patient or use a device to test this.



Do not adjust your television...

You can then single tap to get back to the normal interactive app. This "feature" illustrates what the app might look like to someone with low visual acuity. As you can see, you can't really read any of the text.

Dynamic type is important for the millions of people without perfect vision to be able to actually see and use your app.

A brief review

While the basic support for dynamic type has been available since iOS 7, iOS 11 introduces a number of new features to make it easier to adopt in your app. When you create a view with text, like a label, one of the properties you can set is the font. You can specify a system font, a custom font, or a font style.

If you specify a style, the system will assign the actual font based on the user's preferred content size. There are several styles available, such as "body", "headline" and "footnote". You can use these different styles to create text elements of different sizes that respond to the user's settings.

You don't have to make every text element respond to changes. Usually, you only have to worry about the ones focused on content. You can set the font style in Interface Builder or in code, using `UIFont.preferredFont(forTextStyle:)`.

In the Settings app, under **Display & Brightness**, you'll find a **Text Size** setting. This setting lets you choose between seven different font sizes. Under **General\Accessibility\Larger Text**, the user can turn on a switch that makes five larger sizes available.

Beginning with iOS 10, your code can read the value of the user's setting using the `preferredContentSizeCategory` property on `UIApplication`, or on objects conforming to the `UITraitCollection` protocol. You can also be notified when the user changes this setting while your app is running, using the `UIContentSizeCategoryDidChange` notification, or through `traitCollectionDidChange`.

How dynamic type impacts layout

There are several layout scenarios where you should consider the length of text: displaying user-supplied content, translating text into different languages, and dynamic type.

For example, when you're designing your user interface, you may think that there is plenty of room on the screen to display all of your text on one line. But if the user chooses a very large font, that may no longer be true. This means that you need to think about more constraints than you may be used to.

If you're already in the habit of adding trailing constraints to your labels, congratulations! You're ready to support dynamic type. But if you're not in that habit yet, there's good news: Xcode 9 will help you get in the habit.

In fact, open **Main.storyboard** in the sample project and you'll see a new warning right away: Trailing constraints are missing on some of the views.

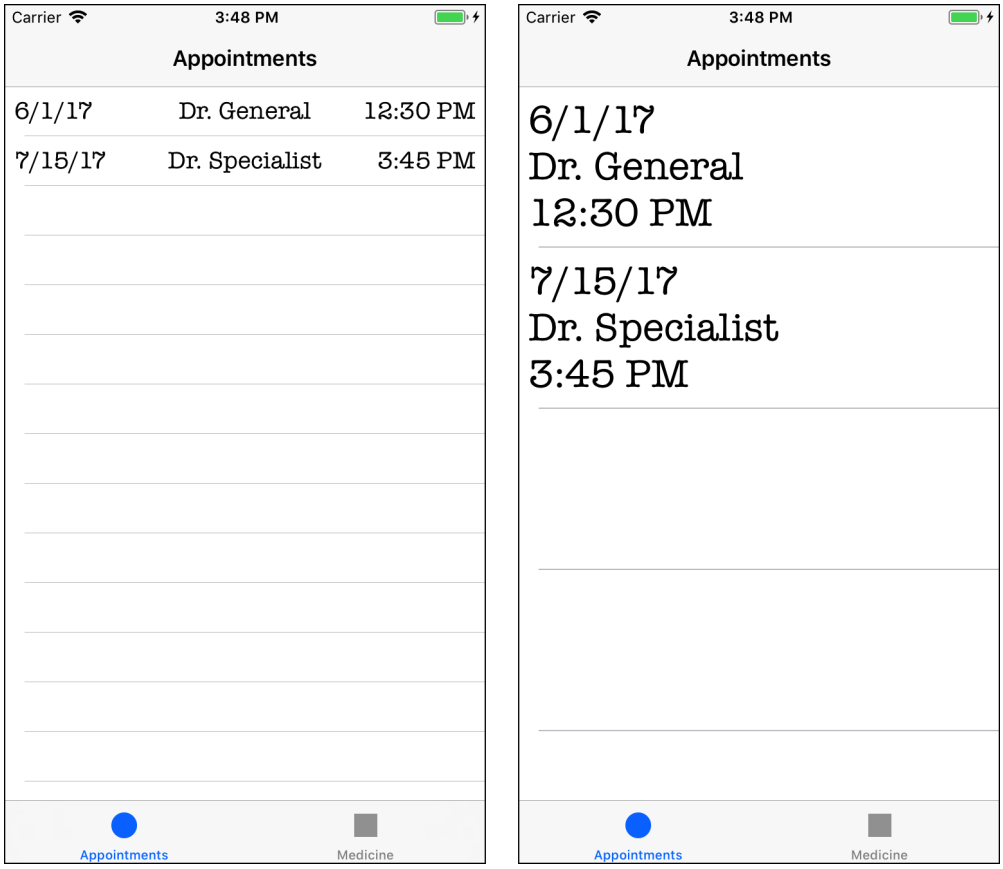
Select the Doctor Scene and add trailing constraints **>= 0** to the superview margin for the **Name**, **Phone**, **Address**, **City/State/ZIP** (CSZ) and **Notes** labels. You should now see all the warnings have cleared.

You may also want to think differently about the number of lines for your label. If your label has number of lines set to 1 and line break set to "truncate tail" (the default settings), two things will happen:

1. Your new trailing constraints will keep the label from running off the right of the screen.
2. The right side will start to truncate the text grows.

That may be what you want, but if you need the text to wrap, then you'll have to set number of lines to **0** and line break to **word wrap**.

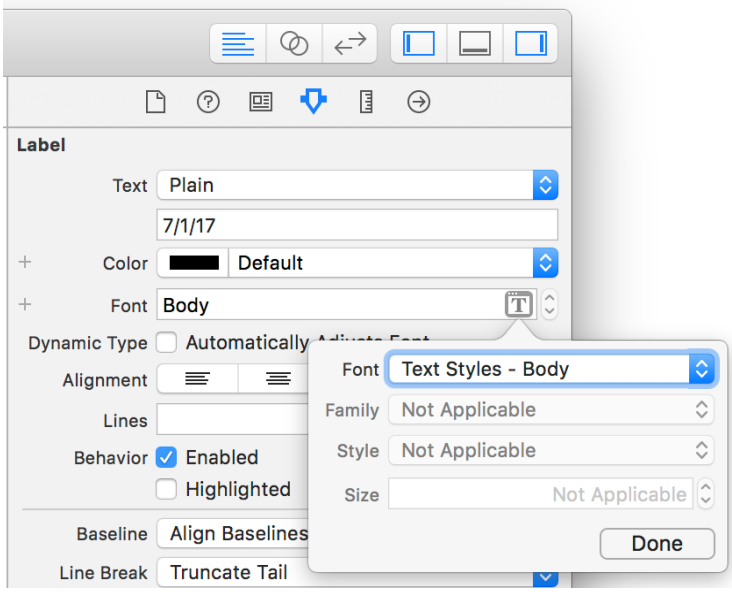
If the setting for text size is very large, it may make sense to adopt a different layout entirely. For example, rather than having text side-by-side, it may make more sense to stack the text vertically. You'll see an example of that later in this chapter.



Changing layout based on text size

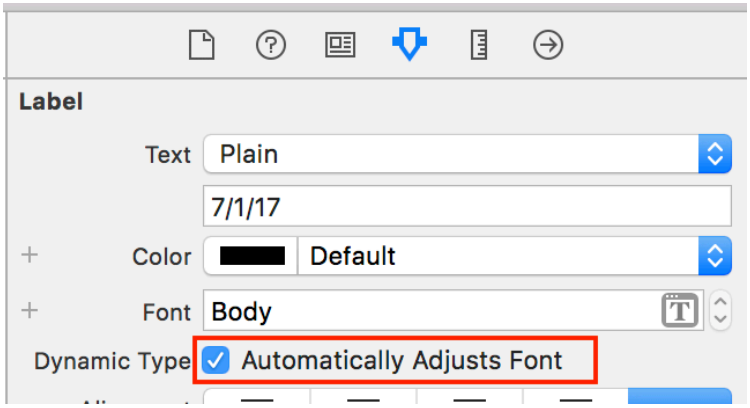
Getting started

Let's convert the starter project to use dynamic type. The first thing you need to do is set a style for each label you want to be dynamic.



In **Main.storyboard**, select the **Appointments Scene** and click on the **Date Label**. In the Attributes inspector, change the font to the **Body** text style.

In Xcode 9, you can instruct the system to adjust the font size automatically when the user changes their preferred size. Just below the Font setting, check the **Automatically Adjusts Font** checkbox:



Repeat both of these steps for the **Doctor Name Label** and the **Time Label**.

Then select the **Medicine Scene** and repeat for the **Title** and **Detail** labels.

Now, select the **Doctor Scene** and change the following:

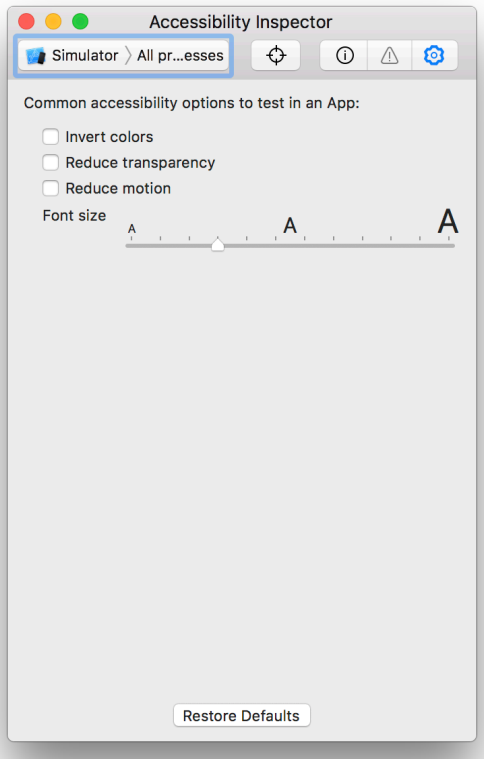
- Name Label style to **Headline**
- Phone, Address, and CSZ Labels to **Footnote**
- Notes Label to **Body**.

Make sure the **Automatically Adjusts Font** checkbox is checked for all of the labels.

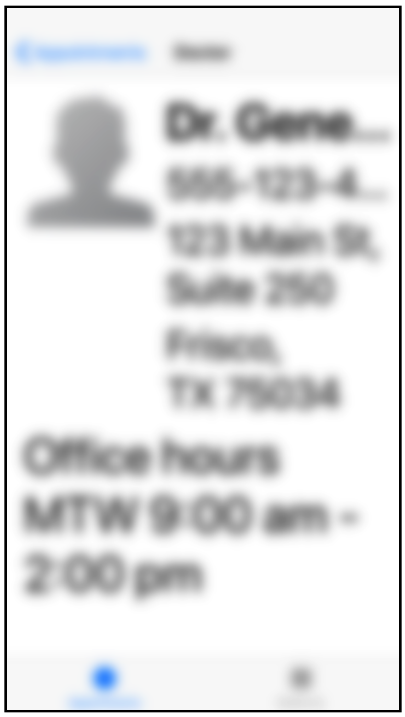
Using the Accessibility Inspector

Build and run the app in the simulator. To change the text size and see your labels adjust, you could background the app, go to the Settings app, navigate to **General\Accessibility\Larger Text**, set a new setting, then switch back to your app — but that's a very tedious way to change the font size!

There's a much better way. From Xcode, click on the **Xcode** menu, then **Open Developer Tool\Accessibility Inspector**. This will bring up a window that lets you view and modify the settings for the simulator.

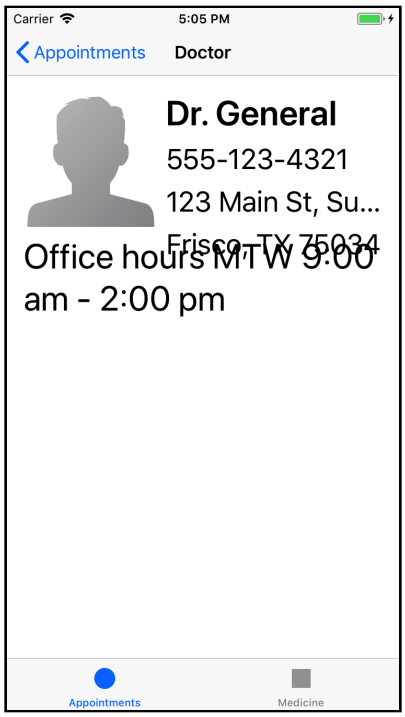


Make sure the simulator is selected and then click on the **Settings** button. From there, you can drag the Font size slider around. When you let go, the size of the text in your app will automatically adjust to the new setting! Try setting the value to one of the larger sizes and then triple-tapping the interface to show the blurry version.



> **Note:** If you're having trouble seeing the font size change in your app, first make sure you're letting go of the slider, as the size won't change as you drag the slider around until you let go. Second, the connection between the simulator and the Accessibility Inspector can sometimes be fragile, so try rebuilding your app after you've opened the Accessibility Inspector. Third, make sure the right instance of the simulator is selected.

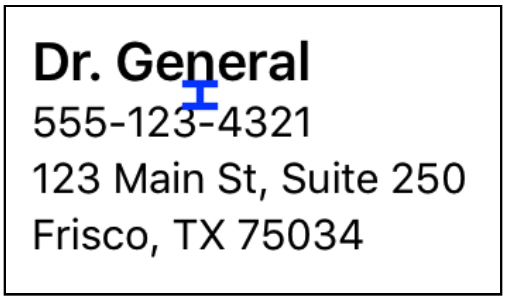
If you tap on a doctor to see the doctor detail view and set the font size to one of the four largest sizes, you'll notice that you have a problem — things start to overlap.



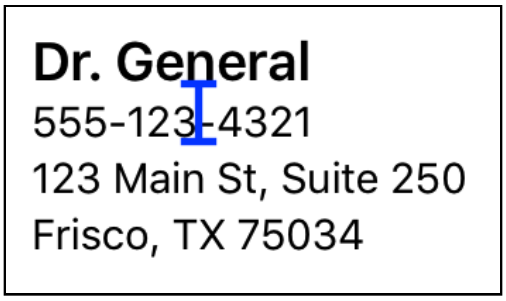
At design time, there is no overlap of the labels, and the Notes label has a constraint to position it below the profile image view. But when the user chooses a font that is large enough, you can see that you'll need some more constraints, specifically between the Notes Label and the CSZ Label.

Baseline to baseline

When creating constraints between labels in a vertical direction, you might create a standard spacing constraint between the bottom of the first label and the top of the second, like so:



This constraint will work, but there is a better way to create constraints between two text elements with standard spacing baseline-to-baseline constraints, like this:



The spacing that looks best between text that is very small, is not the same spacing that will look best between text that is very large. When you create a baseline-to-baseline constraint with standard spacing, the system will take text size into account when determining what that standard spacing should be.

So, before you add your new constraints between the Notes Label and CSZ Label, change the existing top–bottom constraints into baseline–baseline constraints. In the Doctor Scene:

1. Select the **Name Label** and double click on the **Bottom Space to Phone Label** constraint in the Size inspector.
2. Change the **First Item** to **Phone Label.First Baseline**.
3. Change the **Second Item** to **Name Label.Last Baseline**.

If the text spans multiple lines, the constraint will go from the baseline of the last line of text in the Name Label to the baseline of the first line of text in the Phone Label.

Because the Constant of the constraint is already set to Standard, you don't need to to anything more to get the system to position the labels using the best vertical spacing between them. Repeat for the **Phone–Address** and **Address–CSZ** spacing.

You have a somewhat complicated relationship between the Notes label, the Profile Image View, and the CSZ Label. You want the Notes label to be below the Profile Image View, but if the text above it gets too tall, you also want it to always stay below the CSZ label. Here's how you accomplish that:

1. Add a baseline–baseline constraint between the **CSZ label** and the **Notes label** that is **>=** the Standard value.
2. Change the existing **Notes Top Space to Profile Image View** constraint priority to **Low** (250).

3. Add another **Notes Top Space to Profile Image View** constraint, but >= the Standard value.

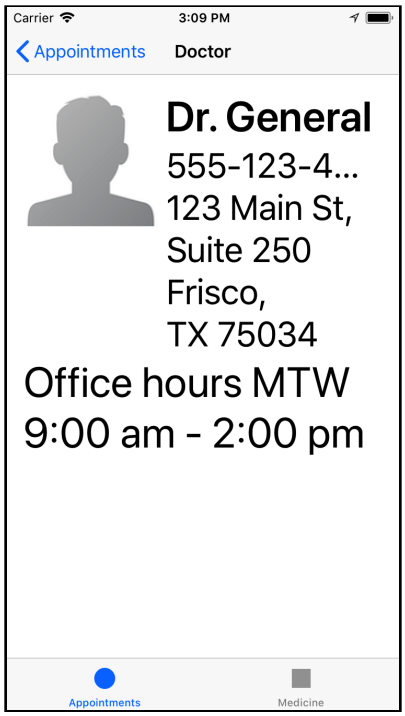
This will keep the Notes label from overlapping the image view, and try to keep it a standard space below it, but allow the notes to move down, if necessary.

There are only a few touch-ups left to do:

1. Increase the **horizontal compression resistance priority** of the **Profile image view** to **751**, to keep it from getting compressed as text gets larger.
2. Change the **Lines** of the **Address** and **CSZ** labels to **0**.
3. Change the **Line Break** of the **Address** and **CSZ** labels to **Word Wrap**.

Notice that you didn't change the Lines and Line Break properties of the Name and Phone Number labels.

Build and run, and you can see how the Phone Number label and the Address label behave differently. Large text will clip at the edge for the phone number, but wrap to multiple lines for the address. Use whichever behavior you need in a particular situation.



UIFontMetrics

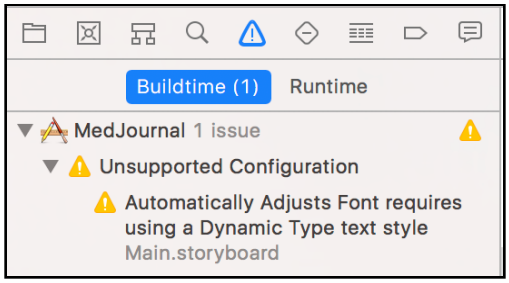
When you choose a font style, you no longer have control over the font used in your app, since the system will choose the font for you. So, if your design calls for a different font, will you find yourself left out of all the Dynamic Type goodness?

Nope! `UIFontMetrics` is a new class that helps scale your custom fonts based on the user's preferred size.

That's not all — `UIFontMetrics` has a couple of other tricks up its sleeve. You can use it to scale numbers, which is useful if you want to adjust the constant of a constraint based on the text size. And you can scale a font while specifying a maximum point size, for cases when your user interface cannot handle text larger than a certain size.

Custom fonts

You'll now change the font of your app to use a custom font. Select the labels in the **Appointments Scene** and change their font to a custom font, such as **American Typewriter 17.0**. At this point, you may notice a new warning:



The Interface Builder checkbox **Automatically Adjusts Font** corresponds to the property `adjustFontForContentSizeCategory`. It's only appropriate to set this value to `true` when using a font style, or when using `UIFontMetrics.scaledFont(for:)`.

Since Interface Builder doesn't know what you're doing with these labels in code, it warns you that you should not set that checkbox to `true` if you're not using a font style. Clear the checkbox for all three labels.

Repeat this process for the labels in the **Medicine Scene**, setting the font to **American Typewriter 17.0** and clearing the **Automatically Adjusts Font** checkbox.

For the **Doctor Scene**, you're going to use a couple of different fonts:

- Set the **Name** label to **American Typewriter Semibold 17.0**.
- Set the **Notes** label to **American Typewriter 20.0**.

- Set the other three to **American Typewriter 13.0**.

Be sure to clear the checkbox on all of those views.

Now that the custom font is set, you'll add the code to make it adjust to the user's preferred size. Open **AppointmentCell.swift** and add the following code before `setupConstraints()`:

```
override func awakeFromNib() {
  super.awakeFromNib()
  // 1
  let metrics = UIFontMetrics(forTextStyle: .body)
  // 2
  dateLabel.font = metrics.scaledFont(for: dateLabel.font)
  doctorNameLabel.font =
    metrics.scaledFont(for: doctorNameLabel.font)
  timeLabel.font = metrics.scaledFont(for: timeLabel.font)

  // 3
  dateLabel.adjustsFontForContentSizeCategory = true
  doctorNameLabel.adjustsFontForContentSizeCategory = true
  timeLabel.adjustsFontForContentSizeCategory = true
}
```

Here's what this is doing:

1. In this case, you can use the same `UIFontMetrics` instance for all the labels, so you create one with the `.body` style.
2. Then you tell each label to use a scaled font using the `UIFontMetrics` instance. You pass the font that was set in the storyboard as the font to scale.
3. Setting this property is the same as setting the checkbox in Interface Builder.

Now, open **DoctorViewController.swift** and add this code:

```
override func awakeFromNib() {
  super.awakeFromNib()
  loadViewIfNeeded()
  nameLabel.font = UIFontMetrics(forTextStyle: .headline)
    .scaledFont(for: nameLabel.font)
  phoneLabel.font = UIFontMetrics(forTextStyle: .footnote)
    .scaledFont(for: phoneLabel.font)
  addressLabel.font = UIFontMetrics(forTextStyle: .footnote)
    .scaledFont(for: addressLabel.font)
  cityStateZipLabel.font =
    UIFontMetrics(forTextStyle: .footnote)
    .scaledFont(for: cityStateZipLabel.font)
  notesLabel.font = UIFontMetrics(forTextStyle: .body)
    .scaledFont(for: notesLabel.font)

  nameLabel.adjustsFontForContentSizeCategory = true
  phoneLabel.adjustsFontForContentSizeCategory = true
```

```
    addressLabel.adjustsFontForContentSizeCategory = true
    cityStateZipLabel.adjustsFontForContentSizeCategory = true
    notesLabel.adjustsFontForContentSizeCategory = true
  }
```

No surprises here; this is simply doing the same thing, and using the right style for each label.

Finally, open **MedicineViewController.swift** and replace these lines in `tableView(_:cellForRowAt:)`:

```
  cell.textLabel?.text = medicine.name
  cell.detailTextLabel?.text = medicine.dose
```

With these:

```
  if let textLabel = cell.textLabel {
    textLabel.text = medicine.name
    textLabel.font = UIFontMetrics(forTextStyle: .body)
      .scaledFont(for: textLabel.font)
    textLabel.adjustsFontForContentSizeCategory = true
  }

  if let detailLabel = cell.detailTextLabel {
    detailLabel.text = medicine.dose
    detailLabel.font = UIFontMetrics(forTextStyle: .body)
      .scaledFont(for: detailLabel.font)
    detailLabel.adjustsFontForContentSizeCategory = true
  }
```

Build and run, and you'll see the interface using your custom font, dynamically adjusted to the correct size.

Scaling sizes

Your little app doesn't have any size values in code, but you can simulate that situation.

Start by opening **Main.storyboard**, selecting the Doctor Scene, and creating an outlet for the width constraint on the Profile Image View. Name the outlet `profileImageSizeConstraint`. Add this constant below the `IBOutlet`:

```
  private let defaultImageSize: CGFloat = 128
```

And add this code to the bottom of the class:

```
  override func traitCollectionDidChange(
    _ previousTraitCollection: UITraitCollection?) {

    // 1
    let preferredSize =
      traitCollection.preferredContentSizeCategory
```

```
    if preferredSize !=
      previousTraitCollection?.preferredContentSizeCategory {

      // 2
      setImageSize()
    }
  }

  private func setImageSize() {
    // 3
    profileImageSizeConstraint.constant =
      UIFontMetrics.default.scaledValue(for: defaultImageSize)
  }
```

Here's what this does:

1. The system calls the view controller's `traitCollectionDidChange` method when the preferred content size category changes. There are several other traits that can change, so you first check that it was this setting that changed.
2. Then you call your `setImageSize` method…
3. …which sets the constant on the constraint to a scaled value, passing the original value to scale. Notice that this uses the `default` instance of `UIFontMetrics`, which simply uses the `.body` font style.

There are two constraint changes you have to make to avoid some constraint errors in the console when running on smaller devices (5s/6/6s/7) and at the largest text settings:

1. Change the priority of the width constraint on the profile image view to 999.
2. Add a trailing space constraint on the profile image view >= 16 to the safe area.

If you build and run, you'll see the size of the profile image view grow as the `UIFontMetrics` class scales the constant for its constraint.

Images

If you want to evenly scale your image based on the text size, you can use the `scaledValue` method on `UIFontMetrics` and a size constraint, as you've just done in the previous section. However, there is an easier way to adjust image sizes for accessibility.

In fact, if you just want your image to be visible for people using the accessibility text size categories, there is a property you can set that will handle the resizing for you.

`UIImageView`'s `adjustsImageSizeForAccessibilityContentSizeCategory` property, when set to `true`, will cause your image size to remain the same for all seven text sizes. But if the user chooses one of the five additional accessibility text sizes, your image will be scaled up to ensure your users can see it.

You can also set this value in Interface Builder, using the **Accessibility\Adjusts Image Size** checkbox in the Attributes inspector for an image view. Try it out:

1. Remove the `traitCollectionDidChange` and `setImageSize` methods from **DoctorViewController.swift**.
2. Open **Main.storyboard** and remove the image view width constraint.
3. Check the **Adjusts Image Size** checkbox

Build and run and change the text size using the Accessibility Inspector to several different values. You should notice the following:

- The image remains the same using any of the smaller text sizes.
- The image is scaled up for the five largest text sizes.

Modifying the layout

It may make sense to adjust the structure of the layout when the user's preferred font size is very large. In fact, you may have already noticed that table cells with a system style will do this for you!

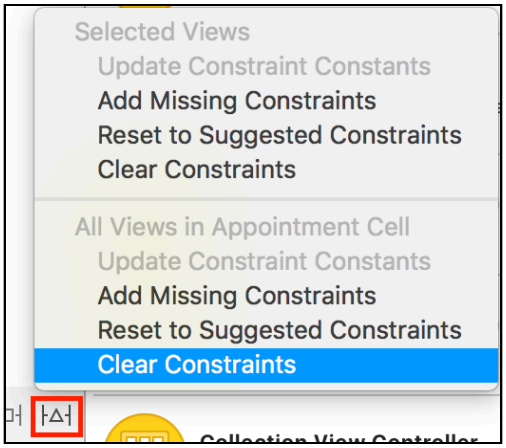
Build and run and view the Medicine tab. Change the font to one of the five largest accessibility sizes and you'll notice that the cell goes from a side-by-side layout to a top-to-bottom layout.

There are a couple of checks that you can do in code to decide whether to adjust the layout:

1. You can check the `preferredContentSizeCategory.isAccessibilityCategory` property of `traitCollection`. This will return `true` if the user has selected one of the five larger accessibility text sizes.
2. You can use comparison operators; `UIContentSizeCategory` implements operators for comparison. For example, you could adjust your layout if `traitCollection.preferredContentSizeCategory > .extraLarge`.

You'll modify the appointment cell to adjust its layout.

Because you're going to set the constraints in code, you'll need to remove them from the storyboard. Open **Main.storyboard**, select one of the labels in the Appointments Scene, and clear all the constraints. To do this, click the triangle in the lower right and choose **Clear Constraints** under **All Views in Appointment Cell**.



You have a `setupConstraints` method in **AppointmentCell.swift** from the starter project, but nothing is calling it. Add this code to `awakeFromNib`:

```
override func awakeFromNib() {
  ...
  setupConstraints()
  updateConstraints()
}
```

`setupConstraints` creates three arrays of constraints: one common to both layouts, one for the regular layout, and one for the layout when larger text is selected.

`updateConstraints` is a `UIView` method that you need to override to activate the correct constraints, so add that now:

```
override func updateConstraints() {
  NSLayoutConstraint.activate(commonConstraints)

  if traitCollection.preferredContentSizeCategory
    .isAccessibilityCategory {

    NSLayoutConstraint.deactivate(regularConstraints)
    NSLayoutConstraint.activate(largeTextConstraints)
  } else {
    NSLayoutConstraint.deactivate(largeTextConstraints)
    NSLayoutConstraint.activate(regularConstraints)
  }

    super.updateConstraints()
}
```

This uses the `isAccessibilityCategory` property to decide which constraints to activate, but you could use comparison operators here instead if you wanted.

Although this will set up the constraints properly when the view is first shown, you also need to check when the preferred size changes. Add the following override to the class:

```
override func traitCollectionDidChange(
  _ previousTraitCollection: UITraitCollection?) {

  let isAccessibilityCategory = traitCollection
    .preferredContentSizeCategory.isAccessibilityCategory

  if isAccessibilityCategory != previousTraitCollection?
    .preferredContentSizeCategory.isAccessibilityCategory {

    setNeedsUpdateConstraints()
  }
}
```

This is similar to what you did before in `traitCollectionDidChange`, but this time you're checking if the `isAccessibilityCategory` value is different and then calling `setNeedsUpdateConstraints`, which will trigger another call to `updateConstraints`.

Build and run, and you'll see the layout change when you pick one of the five larger text sizes.

Where to go from here?

There are a couple of sessions from WWDC 2017 that are helpful for this topic, including one that is a bit of a surprise!

- **Building Apps with Dynamic Type**: You would expect this sesstion to be loaded with good information on this topic, and you won't be disappointed. apple.co/2uSN6qX
- **Localizing with Xcode 9**: There's a little nugget here that shows you how to use a strings dictionary (`stringsdict` file) to create adaptive strings using `NSStringVariableWidthRuleType` and `variantFittingPresentationWidth` to present a completely different string based on the available space. This could be handy for a variety of situations! apple.co/2vuvUvF

Although you've covered quite a lot of territory in this chapter, if you're already using Auto Layout, adding support for dynamic type is really rather simple — and helpful for millions of users!