LocalizeMe App
Application Localization

IOS devices are available in about hundred different countries, and that number will continue to increase over time. You can now buy and use an iPhone on every continent except Antarctica. The iPad and iPod touch are also sold all over the world and are nearly as ubiquitous as the iPhone. If you plan on releasing applications through the App Store, your potential market is considerably larger than just people in your own country who speak your own language. Fortunately, iOS has a robust **localization** architecture that lets you easily translate your application (or have it translated by others) into, not only multiple languages, but even into multiple dialects of the same language. For instance, you may provide a different terminology to English speakers in the United Kingdom than you may do to English speakers in the United States.

That is, localization is no problem if you've written your code correctly. Retrofitting an existing application to support localization is much harder than writing your application that way from the start. In this practice, we'll look how to write a code so it is easy to localize, and then we'll go about localizing a sample application.

Localization Architecture

When a nonlocalized application is run, all of the application's text will be presented in the developer's own language, also known as the **development base language**. When developers decide to localize their applications, they create a subdirectory in their application bundle for each supported language. Each language's subdirectory contains a subset of the application's resources that were translated into that language. Each subdirectory is called a **localization project**, or **localization folder**. Localization folder names always end with the ***.lproj*** extension.

In the iOS Settings application, the user has the ability to set the device's preferred language and region format. For example, if the user's language is English, available regions might be the United States, Australia, and Hong Kong—all regions in which English is spoken.

When a localized application needs to load a resource—such as an image, property list, or nib—the application checks the user's language and region, and then looks for a localization folder that matches that setting. If it finds one, it will load the localized version of the resource instead of the base version. For users who select French as their iOS language and Switzerland as their region, the application will look first for a localization folder named ***fr-CH.lproj***. The first two letters of the folder name are the ISO country code that represents the French language. The two letters following the hyphen are the ISO code that represents Switzerland.

If the application cannot find a match using the two-letter code, it will look for a match using the language's three-letter ISO code. In our example, if the application is unable to find a folder named *fr-CH.lproj*, it will look for a localization folder named *fre-CH* or *fra-CH*.

All languages have at least one three-letter code. Some have two three-letter codes: one for the English spelling of the language and another for the native spelling. Some languages have only two-letter codes. When a language has both a two-letter code and a three-letter code, the two-letter code is preferred.

Note You can find a list of the current ISO country codes on the ISO web site (www.iso.org/iso/ country_codes.htm). Both the two- and three-letter codes are part of the ISO 3166 standard.

If the application cannot find a folder that is an exact match, it will then look for a localization folder in the application bundle that matches just the language code without the region code. So, staying with our French-speaking person from France, the application next looks for a localization folder called *fr.lproj*. If it doesn't find a language folder with that name, it will look for

fre.lproj and then *fra.lproj*. If none of those is found, it checks for *French.lproj*. The last construct exists to support legacy Mac OS X applications; generally speaking, you should avoid it.

If the application doesn't find a language folder that matches either the language/region combination or just the language, it will use the resources from the development base language. If it does find an appropriate localization folder, it will always look there first for any resources that it needs. If you load a UIImage using imageNamed(), for example, the application will look first for an image with the specified name in the localization folder. If it finds one, it will use that image. If it doesn't, it will fall back to the base language resource.

If an application has more than one localization folder that matches — for example, a folder called *fr-CH.lproj* and one called *fr.lproj*—it will look first in the more specific match, which is *fr-CH.lproj* if the user has selected Swiss French as their preferred language. If it doesn't find the resource there, it will look in *fr.lproj*. This gives you the ability to provide resources common to all speakers of a language in one language folder, localizing only those resources that are impacted by differences in dialect or geographic region.

You should choose to localize only those resources that are affected by language or country. For example, if an image in your application has no words and its meaning is universal, there's no need to localize that image.

What do you do about string literals and string constants in your source code?

Consider this source code:

```
let alertController = UIAlertController(title: "Location Manager Error",
                          message: errorType, preferredStyle: .Alert)
let okAction = UIAlertAction(title: "OK", style: .Cancel, handler: nil)
alertController.addAction(okAction)
self.presentViewController(alertController, animated: true, completion: nil)
```

If you've gone through the effort of localizing your application for a particular audience, you certainly don't want to be presenting alerts written in the development base language. The answer is to store these strings in special text files called **strings files**.

What's in a Strings File?

Strings files are nothing more than Unicode text files that contain a list of string pairs, each identified by a comment. Here is an example of what a strings file might look like in your application:

```
/* Used to ask the user his/her first name */
"LABEL_FIRST_NAME" = "First Name";

/* Used to get the user's last name */
"LABEL_LAST_NAME" = "Last Name";

/* Used to ask the user's birth date */
"LABEL_BIRTHDAY" = "Birthday";
```

The values between the /* and the */ characters are just comments for the translator. They are not used in the application, and you could skip adding them, though they're a good idea. The comments give context, showing how a particular string is being used in the application. You'll notice that each line is in two parts, separated by an equals sign. The string on the left side of the equal sign acts as a key, and it will always contain the same value, regardless of language. The value on the right side of the equal sign is the one that is translated to the local language. So, the preceding strings file, localized into French, might look like this:

```
/* Used to ask the user his/her first name */
"LABEL_FIRST_NAME " = "Prénom";
/* Used to get the user's last name */
"LABEL_LAST_NAME" = "Nom de famille";
/* Used to ask the user's birth date */
"LABEL_BIRTHDAY" = "Anniversaire";
```

The Localized String Function

At run time, you'll get the localized versions of the strings that you need by using the NSLocalizedString() function. Once your source code is final and ready for localization, you can have Xcode search all your code files for occurrences of this function, pull out all the unique strings and embed them in a file that you can send to a translator, or you can add the translations yourself. Once that's done, you'll have Xcode import the updated file and use its content to create the localized string files for the languages for which you have provided translations. Let's see how the first part of this process works. First, here's a traditional string declaration:

```
let myString = @"First Name"
```

To make this string localizable, do this instead:

```
let myString = NSLocalizedString("LABEL_FIRST_NAME",
                    comment: "Used to ask the user his/her first name")
```

The NSLocalizedString() macro takes five parameters, but three of them have defaults that are good enough in most cases, so usually you only need to provide two:

- The first parameter is a key that will be used to look for the localized string. If there is no localization that contains text for the key, the application will use the key itself as the localized text.

- The second parameter is used as a comment to explain how the text is being used. The comment will appear in the file sent to the translator and in the localized strings file after import.
  NSLocalizedString() looks in the application bundle inside the appropriate localization folder for a strings file named *Localizable.strings*. If it does not find the file, it returns its first parameter, which is the key for the text that was required. If

NSLocalizedString() finds the strings file, it searches the file for a line that matches its first parameter. In the preceding example, NSLocalizedString() will search the strings file for the string "LABEL_FIRST_NAME". If it doesn't find a match in the localization folder that matches the user's language settings, it will then look for a strings file in the base language and use the value there. If there is no strings file, it will just use the first parameter you passed to the NSLocalizedString() function. You could use the base language text as the key for the NSLocalizedString() function because it returns the key if no matching localized text can be found. This would make the preceding example look like this:

```
let myString = NSLocalizedString("First Name",
                    comment: "Used to ask the user his/her first name")
```

However, this approach is not recommended for two reasons. First, it is unlikely that you will come up with the perfect text for your app on your first try. Going back and changing all keys in the strings files is cumbersome and error-prone, which means that you will most likely end up with keys that do not match what is used in the app, anyway. The second reason is that, by clearly using uppercase keys, you can immediately notice if you have forgotten to localize any text when you run the app just by looking at it.

Now that you have an idea of how the localization architecture and the strings file work, let's take a look at localization in action.

Real-World iOS: Localizing Your Application

We're going to create a small application that displays the user's current **locale**. A locale (an instance of NSLocale) represents both the user's language and region. It is used by the system to determine which language to use when interacting with the user, as well as how to display dates, currency, and time information,

among other things. After we create the application, we will then localize it into other languages. You'll learn how to localize storyboard files, strings files, images, and even your application's display name.

You can see what our application is going to look like in Figure 22-1. The name across
the top comes from the user's locale. The ordinals down the left side of the view are static labels, and their values will be set by localizing the storyboard file. The words down the right side, and the flag image at the bottom of the screen, will all be chosen in our app's code at runtime based on the user's preferred language.

Figure 22-1. *The LocalizeMe application shown with two different language settings* Let's hop right into it.

Setting Up LocalizeMe

Create a new project in Xcode using the Single View Application template and call it *LocalizeMe*. Download a pair of images and

name them as *flag_usa.png* and *flag_france.png*. In Xcode, select the *Assets. xcassets* item, and then drag both *flag_usa.png* and *flag_france.png* into it. Now let's add some label outlets to the project's view controller. We need to create one outlet for the label across the top of the view, another for the image view that will show a flag, and an outlet collection for all the words down the right-hand side (see Figure 22-1). Select *ViewController. swift* and make the following changes:

```
class ViewController: UIViewController {
    @IBOutlet var localeLabel : UILabel!
    @IBOutlet var flagImageView : UIImageView!
    @IBOutlet var labels : [UILabel]!
```

Now select *Main.storyboard* to edit the GUI in Interface Builder. Drag a label from the library, dropping it at the top of the main view, aligned with the top blue guideline. Resize the label so that it takes the entire width of the view, from the left margin guidline to the right margin guideline. With the label selected, open the Attributes Inspector. Look for the **Font** control and click the small **T** icon it contains to bring up a small font-selection pop-up. Click the **Style** selector and change it to **Bold** to make the title label stand out a bit from the rest. Next, use the Attributes Inspector to set the text alignment to centered. You can also use the font selector to make the font size larger if you wish. As long as **Autoshrink** is set to **Minimum Font Size** in the object Attributes Inspector, the text will be resized if it gets too long to fit. With your label in place, Control-drag from the **View Controller** icon in the Document Outline (or the one in the storyboard) to this new label, and then select the **localeLabel** outlet.

Next, drag five more labels from the library and put them against the left margin using the blue guideline, one above the other (again, see Figure 22-1). Double-click the top one and change its text from *Label* to *First*. Repeat this procedure with the other four labels, changing the text to the words *Second*, *Third*, *Fourth*, and *Fifth*. Make sure that all five labels are alaigned to the left margin guideline.

Drag another five labels from the library, this time placing them against the right margin. Change the text alignment using the object Attributes Inspector so that they are right- aligned. Control-drag from **View Controller** to each of the five new labels, connecting each one to the labels outlet collection, and making sure to connect them in the right order from top to bottom.

Drag an **Image View** from the library over to the bottom part of the view, so that it touches the bottom and left blue guidelines. In the Attributes Inspector, select **flag_usa** for the view's Image attribute and resize the image horizontally to stretch from blue guideline to blue guideline, and vertically so that it is about a third of the height of the user interface. In the Attributes Inspector, change the Mode attribute from its current value to *Aspect Fit*. Not all flags have the same aspect ratio, and we want to make sure the localized versions of the image look right. Selecting this option will cause the image view to resize any images that it displays so they fit, but it will also maintain the correct aspect ratio (ratio of height to width). Now Control-drag from the view controller to this image view and select the **flagImageView** outlet.

To complete the user interface, we need to set the auto layout constraints. Starting with the label at the top, **Control**-drag from it to its parent view in the Document Outline, press the **Shift** key, select **Leading Space to Container Margin**, **Trailing Space to Container Margin**, and **Vertical Spacing to Top Layout Guide**, and press **return**.

Next, we'll fix the positions of each of the five rows of labels. Control-drag from the label with the text *First* to its parent view in the Document Outline, select both **Leading Space to Container Margin** and **Vertical Spacing to Top Layout Guide**, and press **return. Control**- drag from the label to the label on the same row to its right and select **Baseline**, and then **Control**-drag from the label on the right to its parent view in the Document Outline and select **Trailing Space to Container Margin**.

You have now positioned the top row of labels. Do exactly the same thing for the other four rows. Next, select all of the five labels on the right by holding down the **Shift** key while clicking them with the mouse, and then **Editor** -> **Size to Fit Content**. Finally, clear the text from all of these labels because we will be setting it from our code.

To fix the position and size of the flag, **Control**-drag from the flag label to its parent view in the Document Outline, select **Leading Space to Container Margin**, **Trailing Space to Container Margin**, and **Vertical Spacing to Bottom Layout Guide**, and press **return**. With the flag label still selected, click the **Pin** button, check the **Height** check box in the pop-up, and press **Add 1 Constraint**. You have now added all of the auto layout constraints that we need.

Save your storyboard, and then switch to *ViewController.swift* and add the following code to the viewDidLoad() method:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a nib.
    let locale = NSLocale.currentLocale()
    let currentLangID = NSLocale.preferredLanguages()[0]
    let displayLang = locale.displayNameForKey(NSLocaleLanguageCode,
value: currentLangID) let capitalized =
displayLang?.capitalizedStringWithLocale(locale)
     localeLabel.text = capitalized

    labels[0].text = NSLocalizedString("LABEL_ONE", comment: "The
number 1")
    labels[1].text = NSLocalizedString("LABEL_TWO", comment: "The
number 2")
    labels[2].text = NSLocalizedString("LABEL_THREE", comment: "The
number 3")
    labels[3].text = NSLocalizedString("LABEL_FOUR", comment: "The
number 4")
    labels[4].text = NSLocalizedString("LABEL_FIVE", comment: "The
number 5")
```

```
    let flagFile = NSLocalizedString("FLAG_FILE", comment: "Name of the
flag")
    flagImageView.image = UIImage(named: flagFile)
}
```

The first thing we do in this code is get an NSLocale instance that represents the user's current locale. This instance tells us both the user's language and region preferences, as set in the device's Settings application:

```
let locale = NSLocale.currentLocale()
```

Next, we grab the user's preferred language. This gives us a two-character code, such as "en" or "fr", or a string like "fr_CH" for a regional language variant:

```
let currentLangID = NSLocale.preferredLanguages()[0]
```

The next line of code might need a bit of explanation:

```
let displayLang = locale.displayNameForKey(NSLocaleLanguageCode, value:
currentLangID)
```

NSLocale works somewhat like a dictionary. It can give you a whole bunch of information about the current user's locale, including the name of the currency and the expected date format. You can find a complete list of the information that you can retrieve in the NSLocale API reference. Here, we're using a method called displayNameForKey(_:value:) to retrieve the actual name of the chosen language, translated into the language of the current locale itself. The purpose of this method is to return the value of the item we've requested in a specific language.

The display name for the French language, for example, is *français* in French, but *French* in English. This method gives you the ability to retrieve data about any locale, so that it can be displayed appropriately for all users. In this case, we want the display name of the user's preferred language in the language currently being used, which is why we pass currentLangID as the second argument.

This string is a two-letter language code, similar to the one we used earlier to create our language projects. For an English speaker, it would be *en*; and for a French speaker, it would be *fr.*

The name we get back from this call is going to be something like "English" or "français"— and it will only be capitalized if language names are always capitalized in the user's preferred language. That's the case in English, but not so in French. We want the name capitalized for displaying as a title, however. Fortunately, NSString has methods for capitalizing strings, including one that will capitalize a string according to the rules of a given locale! Let's use that to turn "français" into "Français":

```
let capitalized = displayLang?.capitalizedStringWithLocale(locale)
```

Here, we are using the fact that the Objective-C class NSString and Swift's String are transparently bridged to call the capitalizedStringWithLocale() method of NSString on our String instance. Once we have the display name, we use it to set the top label in the view:

```
localeLabel.text = capitalized
```

Next, we set the five other labels to the numbers 1 through 5, spelled out in our development base language. We use the NSLocalizedString() function to get the text for these labels, passing it the key and a comment indicating what each word is. You can just pass an empty string for the comment if the words are obvious, as they are here; however, any string you pass in the second argument will be turned into a comment in the strings file, so you can use this comment to communicate with the person doing your translations:

```
labels[0].text = NSLocalizedString("LABEL_ONE", comment: "The number 1")
labels[1].text = NSLocalizedString("LABEL_TWO", comment: "The number 2")
labels[2].text = NSLocalizedString("LABEL_THREE", comment: "The number
3")
```

```
labels[3].text = NSLocalizedString("LABEL_FOUR", comment: "The number
4")
labels[4].text = NSLocalizedString("LABEL_FIVE", comment: "The number 5”)
```

Finally, we do another string lookup to find the name of the flag image to use and populate our image view with the named image:

```
let flagFile = NSLocalizedString("FLAG_FILE", comment: "Name of the flag")
flagImageView.image = UIImage(named: flagFile)
```

Let’s run our application now.

Trying Out LocalizeMe

You can use either the simulator or a device to test LocalizeMe. Once the application launches, it should look like Figure 22-2.

Figure 22-2. *The language running under the authors’ base language. The application is set up for localization, but it is not yet localized*

Because we used the NSLocalizedString() function instead of static strings, we are now ready for localization. However, we are not localized yet, as is glaringly obvious from the uppercase labels in the right column and the lack of a flag image at the bottom. If you use the Settings application on the simulator or on your iOS device to change to another language or region, the results look essentially the same, except for the label at the top of the view (see Figure 22-3). If you're not sure how to change the language, hold off for now and I'll explain in detail later, when there is something worth seeing.

Figure 22-3. The nonlocalized application running on an iPhone and set to use the French language

Localizing the Project

Now let's localize the project. In Xcode's Project Navigator, single-click **LocalizeMe**, click the **LocalizeMe** project (not one of the targets) in the editing area, and then select the **Info** tab for the project. Look for the Localizations section in the **Info** tab. You'll see

that it shows one localization, which is for your development language—in my case, that's English. This localization is usually referred to as the **base** localization and it's added automatically when Xcode creates a project. We want to add French, so click the plus (**+**) button at the bottom of the Localizations section and select **French (fr)** from the pop-up list that appears (see Figure 22-4).

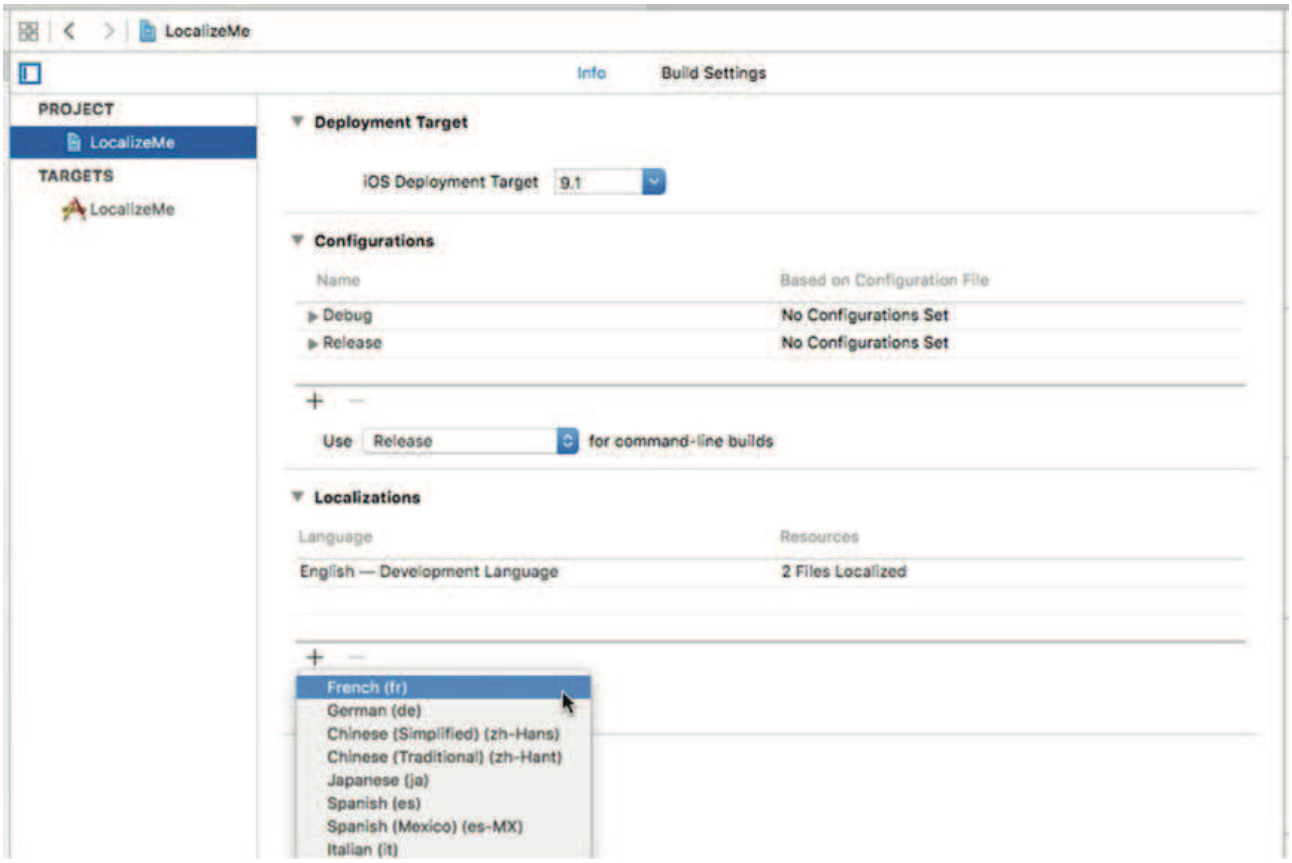

Figure 22-4. The project info settings showing localizations and other information

Next, you will be asked to choose all existing localizable files that you want to localize and which existing localization you want the new French localization to start from (see Figure 22-5). Sometimes when you add a new language, it is advantageous to start with the files for the new language based on those for another one for which you already have a localization—for example, to create a

Swiss French localization in a project that's already been translated into French (as we will, later in this practice), you would almost certainly prefer to use the existing French localization as the start point instead of your base language, and you would do this by selecting **French** as the Reference Language when you add the Swiss French localization. Right now, though, there are only two files to be localized and one choice of starting point language (your base language), so just leave everything as it is and click **Finish**.

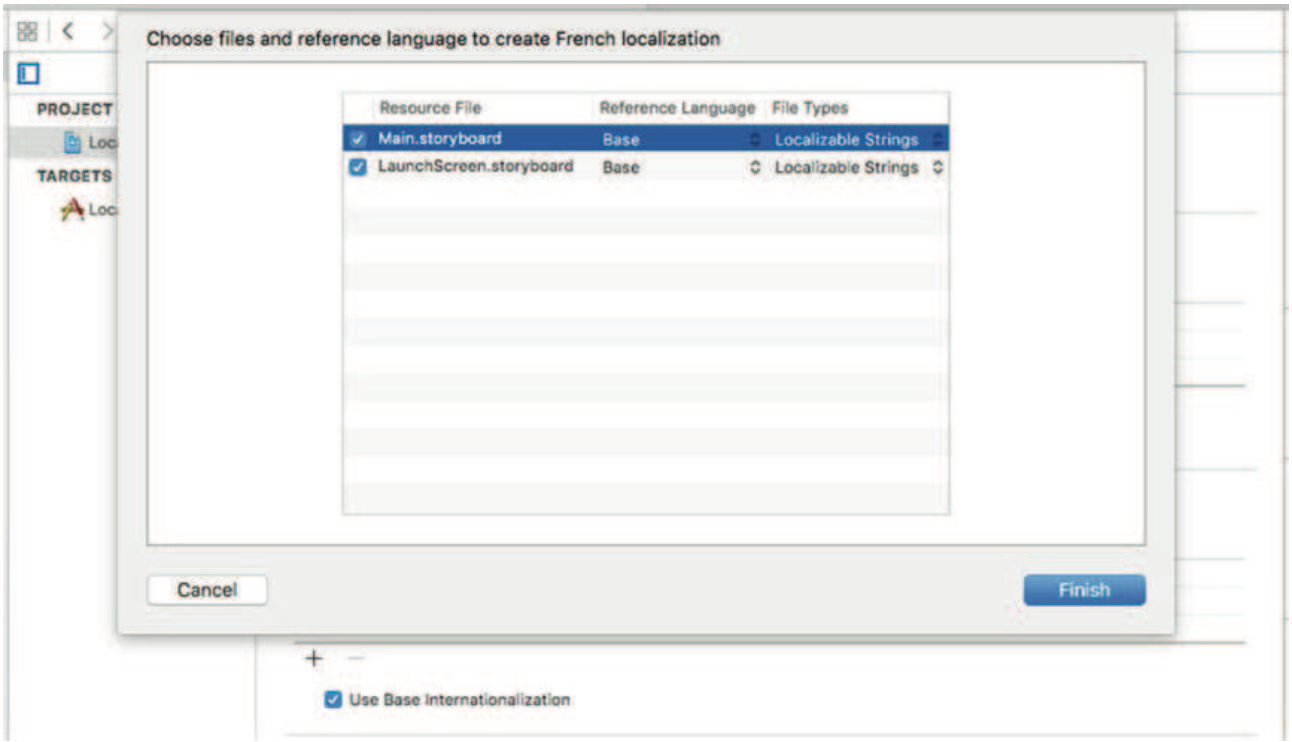

Figure 22-5. *Choosing the files for localization*

Now that you've added a French localization, take a look at the Project Navigator. Notice that the *Main.storyboard* and *Launch.storyboard* files. now have a disclosure triangle next to them, as if they were a group or folder. Expand them and take a look (see Figure 22-6).

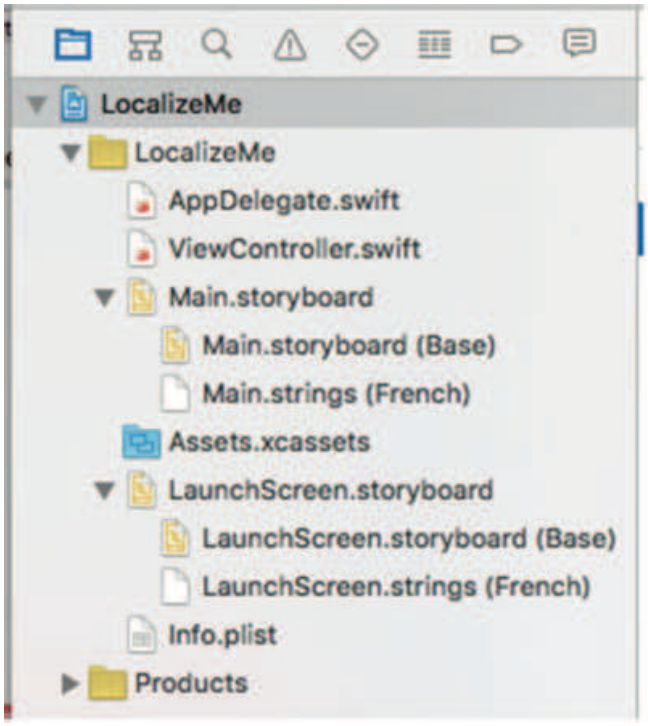

Figure 22-6. *Localizable files have a disclosure triangle and a child value for each language or region you add*

In our project, *Main.storyboard* is now shown as a group containing two children. The first is called *Main.Storyboard* and tagged as *Base*; the second is called *Main.strings* and tagged as *French*. The Base version was created automatically when you created the project, and it represents your development base language. The same applies to the *LaunchScreen. storyboard* file. These files actually live in two different folders: one called *Base.lproj* and one called *fr.lproj*. Go to the Finder and open the *LocalizeMe* folder within your *LocalizeMe* project folder. In addition to all your project files, you should see folders named *Base.lproj* and *fr.lproj* (see Figure 22-7).

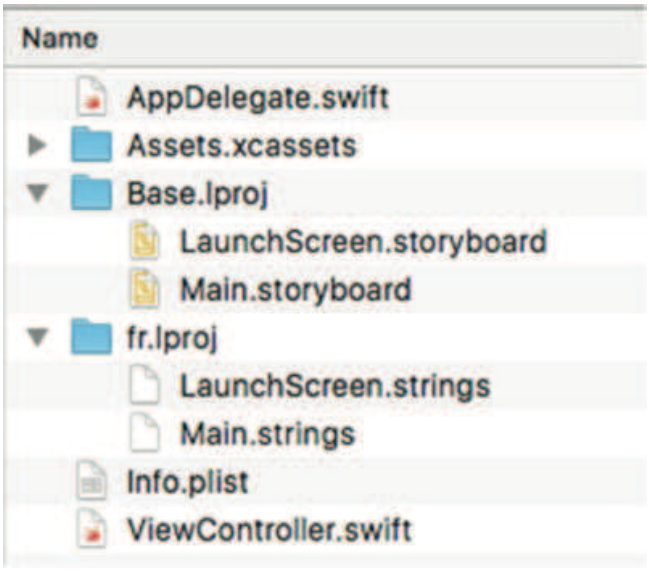

Figure 22-7. *From the outset, our Xcode project included a Base language project folder (Base.lproj). When we chose to make a file localizable, Xcode created a language folder (fr.lproj) for the language we selected*

Note that the *Base.lproj* folder was there all along, with its copies of *Main.storyboard* and *LaunchScreen.storyboard* inside it. When Xcode finds a resource that has exactly one localized version, it displays it as a single item. As soon as a file has two or more localized versions, Xcode displays them as a group. When you asked Xcode to create the French localization, it created a new localization folder in your project called *fr.lproj* and placed in it strings files that contain values extracted from *Base.lproj/ Main.storyboard and Base.lproj/ LaunchScreen.storyboard*. Instead of duplicating both files, Xcode just extracts every text string from them and creates strings files ready for localization. When the app is compiled and run, the values in the localized strings files are pulled in to replace the values in the storyboard and launch screen.

Localizing the Storyboard

In Xcode's Project Navigator, select **Main.strings (French)** to open the French strings file, the contents of which will be injected into

the storyboard shown to French speakers. You'll see something like the following text:

```
/* Class = "UILabel"; text = "Fifth"; ObjectID = "5tN-O9-txB"; */
"5tN-O9-txB.text" = "Fifth";

/* Class = "UILabel"; text = "Third"; ObjectID = "GO5-hd-zou"; */
"GO5-hd-zou.text" = "Third";
/* Class = "UILabel"; text = "Second"; ObjectID = "NCJ-hT-XgS"; */
"NCJ-hT-XgS.text" = "Second";
/* Class = "UILabel"; text = "Fourth"; ObjectID = "Z6w-b0-U06"; */
"Z6w-b0-U06.text" = "Fourth";
/* Class = "UILabel"; text = "First"; ObjectID = "kS9-Wx-xgy"; */
"kS9-Wx-xgy.text" = "First";
/* Class = "UILabel"; text = "Label"; ObjectID = "yGf-tY-SVz"; */
"yGf-tY-SVz.text" = "Label";
```

Each of the pairs of lines represents a string that was found in the storyboard. The comment tells you the class of the object that contained the string, the original string itself, and a unique identifier for each object (which will probably be different in your copy of this file). The line after the comment is where you actually want to add the translated string on the right-hand side of the equals sign. You'll see that some of these are ordinals such as *First*; those come from the labels on the left of Figure 22-3, all of which were given names in the storyboard. The entry with the name *Label* is for the title label, which we set programmatically, so you don't need to localize it.

Prior to iOS 8, the usual practice was to localize the storyboard by directly editing this file. With iOS 8, you can still do that if you choose, but if you plan to use a professional translator, it's likely to be more convenient to have them translate the storyboard text and the strings in your code at the same time. For that reason, Apple has made it possible to collect all of the strings that need to be translated into one file per language that you can send to your translator. If you plan to use that approach, you would leave the storyboard strings file alone and proceed to the next step, which is

described in the next section. It's still possible to modify the storyboard strings file and, if you do so, those changes would not be lost should you need to have your translator make changes or localize additional text. So, just on this occasion, let's localize the storyboard strings in the old-fashioned way. To do so, locate the text for the labels *First*, *Second*, *Third*, *Fourth*, and *Fifth* and then change the string to the right of the equal sign to *Premier*, *Deuxième*, *Troisième*, *Quatrième*, and *Cinquième*, respectively. Finally, save the file.

Your storyboard is now localized in French. There are three ways to see the effects of this localization on your application—you can preview it in Xcode, use a customized scheme to launch it, or change the active language on the simulator or a real device. Let's look at these options in turn, starting with getting a preview.

Using the Assistant Editor to Preview Localizations

Select *Main.storyboard* in the Project Navigator and open the Assistant Editor. In the Assistant Editor's jump bar, select **Preview** -> **Main.storyboard** and you'll see the application as it appears in the base development language, as shown on the left in Figure 22-8.

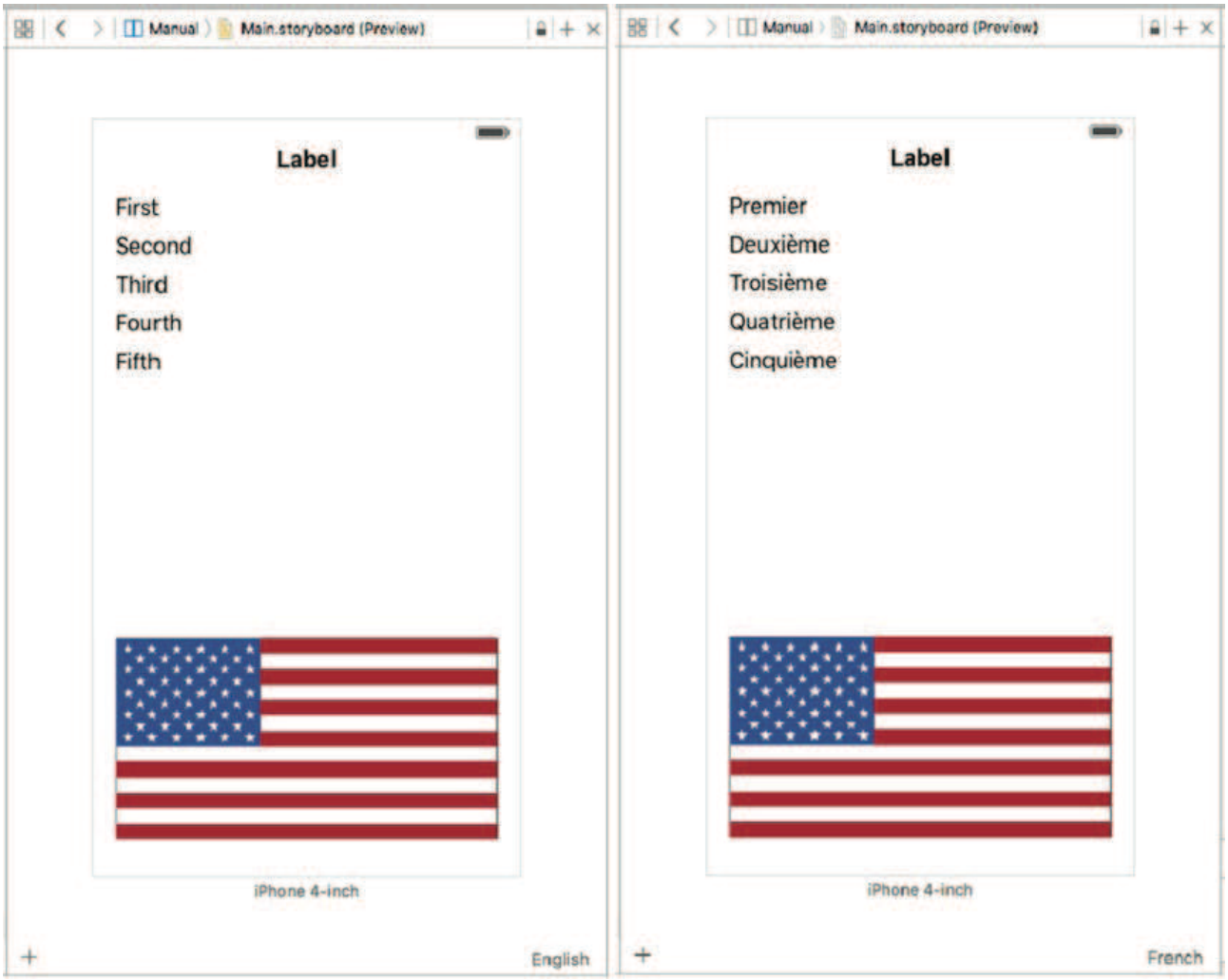

Figure 22-8. *Previewing the application in the base language and in French*

At the bottom right of the Assistant Editor, you'll see that the current language (English) is displayed. Click here to open a pop-up that lists all of the localizations in your project and select **French**. The preview updates to show how the application appears to a French user, as shown on the right in Figure 22-8, except that the flag is not correct. That's because the preview considers only what's in the localized version of the storyboard, whereas we are actually setting the flag image in code. If you're using code to install localized resources, you'll need to choose one of the other options to get an accurate view.

Using a Custom Scheme to Change Language and Region Settings

Creating a customized scheme gives you a quick way to see a localized version of your application running on the simulator or on a real device. Unlike preview, this options lets
you see localizations made in code as well as in the storyboard. Start by clicking on the left side of the Scheme selector in Xcode—you'll find it in the top bar, next to the **Run** and **Stop** buttons. Currently, the selector should be displaying the text **LocalizeMe**, which is the name of the current Scheme, and the currently selected device or simulator. When you click on **LocalizeMe**, Xcode opens a pop-up with several options. Chose **Manage Schemes...** to open the Scheme dialog (see Figure 22-9).

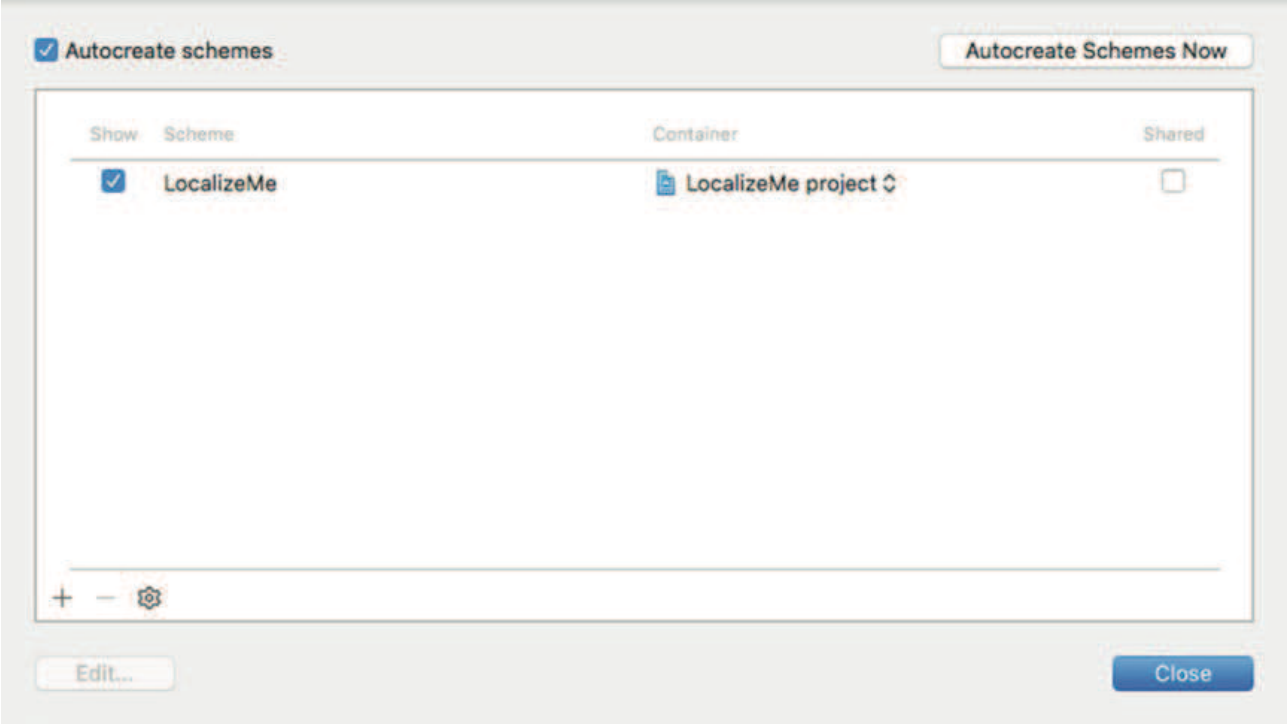

Figure 22-9. *The Scheme dialog lets you view, add and remove schemes*

Currently, there is only one scheme. Click the **+** icon below the scheme list to open another window that let's you choose a name for your new scheme. Call it LocalizeMe_fr and press **OK**. Back in the Scheme dialog, select your newly created scheme and click **Edit...** to open the Scheme editor (see Figure 22-10).

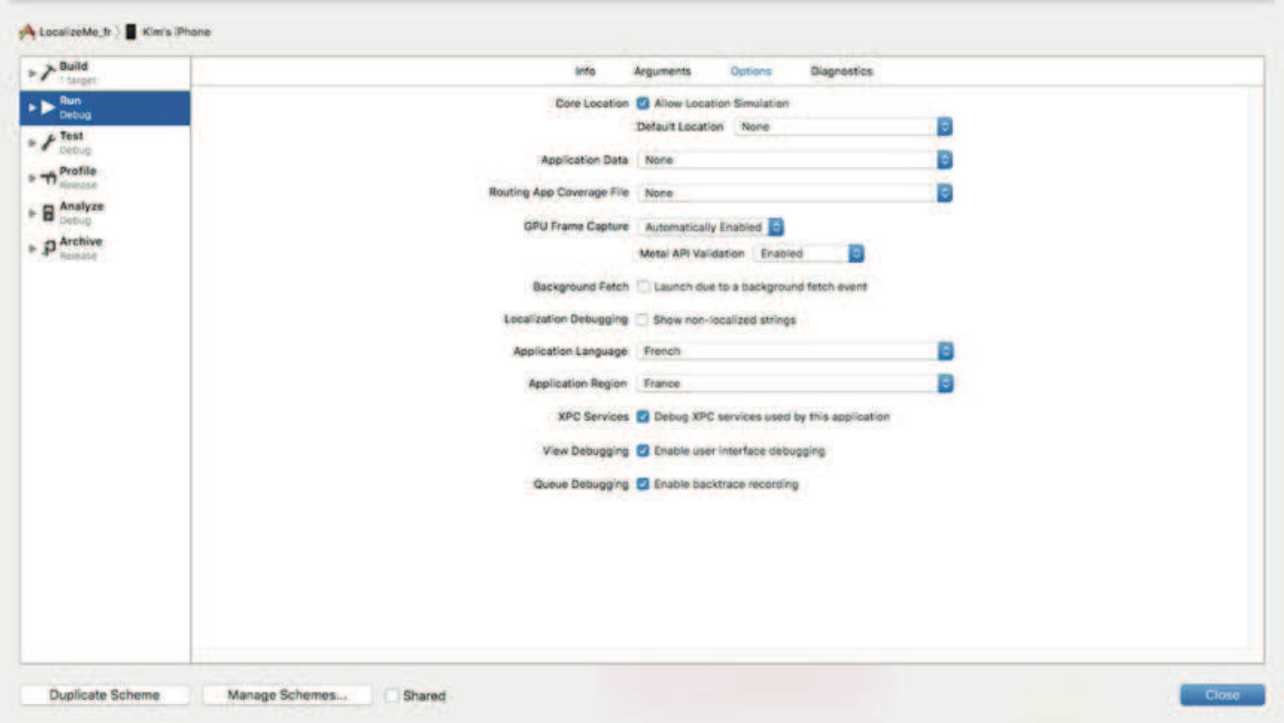

Figure 22-10. *The Scheme editor, with French selected as the Application Language and France as the Application Region*

Make sure that **Run** is selected in the left column and then turn your attention to the **Application Language** and **Application Region** selectors in main body of the editor. Here, you can choose the language and region to be used when the application is launched with your custom scheme. Choose French as the language and France as the region, then click **Close**. Back in Xcode's main window, you'll see that your new scheme is now

selected. Go ahead and run the application and you'll see that the French localization is active, as shown in Figure 22-11.

Figure 22-11. *Viewing the current state of the application in French*

As you can see, the flag is missing. Of course, that's because we're installing the flag image in code and we haven't completed the French localization yet. To revert to the base language view, just switch back to the original **LocalizeMe** scheme and run the application again.

Switching the Language and Region Settings on a Device or Simulator

The final way to see how the application looks in a different language or with different regional settings is to switch those settings in the simulator or on your device. This takes a little more time than either of the other two options, so it's probably better to do this only at the end of the testing cycle, when you are pretty sure that everything works. Here's how to make French the primary language for your device (or simulator).

Go to the Settings application, select the **General** row, and then select the row labeled **Language and Region**. From here, you'll be able to change your language and region preferences (see Figure 22-12).

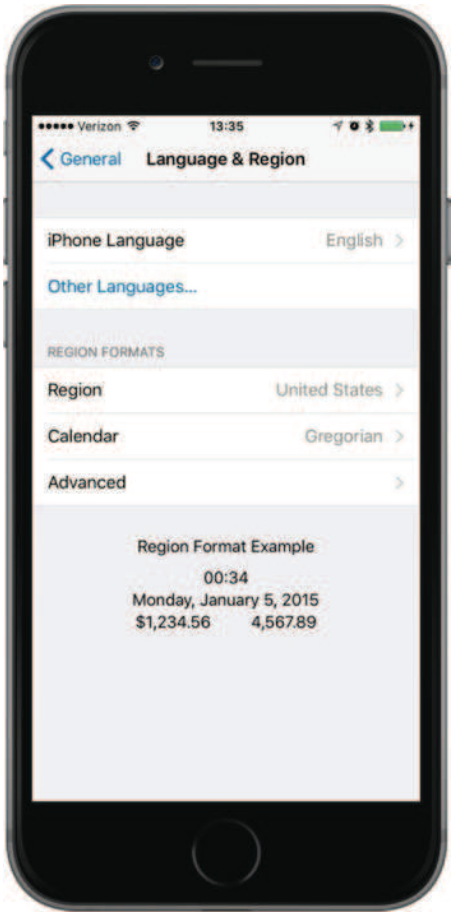

Figure 22-12. *Changing the language or region setting*

Touch **iPhone Language** to reveal the list of languages into which iOS has been localized, and then find and select the entry for

French (which appears in French, as *Français*). You can also change the Region to France for complete authenticity, although that's not necessary for this example since we're not using numbers, dates or times in our code. Press **Done** and then confirm that you want the device language to be changed. This will cause the device to do a partial reboot, which will take a few seconds. Now run the app again and you'll see once again that the labels on the left side are showing the localized French text (see Figure 22-11). However, as before, the flag is missing and right-hand column of text is still wrong. We'll take care of those in the next section.

Generating and Localizing a Strings File

In Figure 22-11, the words on the right side of the view are still in SHOUT_ALL_CAPS style because we haven't translated them yet; what you are seeing are the keys that NSLocalizedString() is using to look for the localized texts. In order to localize those, we need to first extract the key and comment strings from the code. Fortunately, Xcode makes it easy to extract the text that needs to be localized from your project and put it all in a separate file for each language. Let's see how that works.

In the Project Navigator, select your project then, in the editor, select either the project or one of its targets. Now choose **Editor -> Export for Localization...** from the menu. This opens a dialog where you choose which languages you want to localize and where the files for each language should be written. Select a suitable location for the file (for example, in a new folder called *XLIFF* in the project's root directory), ensure that **Existing Translations** and the check box for **French** are both selected and press **Save**. Xcode will create a file called *fr.xliff* in a folder called LocalizeMe inside the folder that you chose. If you plan to use a third-party service to translate your application's text, it's likely that they can work with XLIFF files—all you should need to do is send them this file, have them update it with the translated strings and

re-import it into Xcode. For now, though, we are going to do the translation ourselves.

Open the *fr.xliff* file. You'll see that it contains a lot of XML. It breaks down into three different sections that contain the strings from the storyboard, the strings that Xcode found in your source code, and a number of localizable values from your application's *Info.plist* file. We'll talk about why you need to localize entries from *Info.plist* later in the practice. For now, let's translate the text that comes from the application's code. Look through the file and you'll find that text embedded in some XML that looks like this:

```
<file original="LocalizeMe/Localizable.strings" source-language="en"
datatype="plaintext"
target-language="fr">
  <header>
    <tool tool-id="com.apple.dt.xcode" tool-name="Xcode" tool-version="7.1.1"
build-
    num="7B1005"/>
  </header>
  <body>
    <trans-unit id="FLAG_FILE">
      <source>FLAG_FILE</source>
      <note>Name of the flag</note>
    </trans-unit>
    <trans-unit id="LABEL_FIVE">
      <source>LABEL_FIVE</source>
      <note>The number 5</note>
    </trans-unit>
    <trans-unit id="LABEL_FOUR">
      <source>LABEL_FOUR</source>
      <note>The number 4</note>
    </trans-unit>
    <trans-unit id="LABEL_ONE">
      <source>LABEL_ONE</source>
      <note>The number 1</note>
    </trans-unit>
    <trans-unit id="LABEL_THREE">
```

```
      <source>LABEL_THREE</source>
      <note>The number 3</note>
    </trans-unit>


    <trans-unit id="LABEL_TWO">
      <source>LABEL_TWO</source>
      <note>The number 2</note>
    </trans-unit>
  </body>
</file>
```

Notice that the <file> element has a target-language attribute that gives the language into which the text needs to be translated and that there is a nested <trans-unit> element for each string that needs to be translated. Each of them contains a <source> element with the original text and a <note> element that contains the comment from the NSLocalizedString() call in the source code. Professional translators have software tools that present the information in this file and allow them to enter the translations. We, on the other hand, are going to do it manually by adding <target> elements containing the French text, like this:

```
<file original="LocalizeMe/Localizable.strings" source-language="en"
datatype="plaintext"
target-language="fr">
  <header>
    <tool tool-id="com.apple.dt.xcode" tool-name="Xcode" tool-version="7.1.1"
build-
    num="7B1005"/>
  </header>
  <body>
<trans-unit id="FLAG_FILE"> <source>FLAG_FILE</source> <note>Name of
the flag</note> <target>flag_france</target>

    </trans-unit>
    <trans-unit id="LABEL_FIVE">
```

<source>LABEL_FIVE</source> <note>The number 5</note> <target>Cinq</target>

</trans-unit>
<trans-unit id="LABEL_FOUR">
<source>LABEL_FOUR</source> <note>The number 4</note>
<target>Quatre</target>

</trans-unit>
<trans-unit id="LABEL_ONE">
<source>LABEL_ONE</source> <note>The number 1</note> <target>Un</target>

</trans-unit>
<trans-unit id="LABEL_THREE">
<source>LABEL_THREE</source> <note>The number 3</note>
<target>Trois</target>

</trans-unit>
<trans-unit id="LABEL_TWO"> <source>LABEL_TWO</source> <note>The number 2</note> <target>Deux</target>

</trans-unit>
</body>
</file>

If you haven't already translated the storyboard strings, you can do that too. You'll find them in a separate block of <trans-unit> elements, which are easy to find because of the comments that include the links to the labels from which the text came. On the other hand, if you have done the translations already, you'll find that Xcode included them in the XLIFF file, like this:

<trans-unit id="GO5-hd-zou.text">
<source>Third</source>
<target>Troisième</target>
<note>Class = "UILabel"; text = "Third"; ObjectID = "GO5-hd-zou";</note>

</trans-unit>
<trans-unit id="NCJ-hT-XgS.text">

```
  <source>Second</source>
  <target>Deuxième</target>
  <note>Class = "UILabel"; text = "Second"; ObjectID = "NCJ-hT-XgS";</note>
</trans-unit>
```

Save your translations. The next step is to import the results back into Xcode. Make sure that the project is selected in the Project Navigator then choose **Editor -> Import Localizations** in the menu bar, navigate to your file, and open it. Xcode will show you a list of keys that have been translated and their translations. Press **Import** to complete the import process. If you at the Project Navigator, you'll see that two files have been added—*InfoPlist. strings* and *Localizable.strings*. Open *Localizable.strings* and you'll see that it contains the French translations of the strings that Xcode extracted from *ViewController.swift*:

```
/* Name of the flag */
"FLAG_FILE" = "flag_france";
/* The number 5 */
"LABEL_FIVE" = "Cinq";
/* The number 4 */
"LABEL_FOUR" = "Quatre";
/* The number 1 */
"LABEL_ONE" = "Un";
/* The number 3 */
"LABEL_THREE" = "Trois";
/* The number 2 */
"LABEL_TWO" = "Deux";
```

Now compile, and run the app with French as the active language. You should see the labels on the right-hand side translated into French (see Figure 22-1); and at the bottom of the screen, you should now see the French flag, as shown on the right in Figure 22-1.

So are we done yet? Not quite. Rerun the app with English as the active language. You'll see the unlocalized version of the app shown in Figure 22-2. To make the app work in English, we have to localize it for English. To do that, select the Project in the Project

Navigator and then select **Editor** -> **Export for Localization...** from the menu to export strings for localization again, but this time choose **Development Language Only**, and then press **Save**. This creates a file called *en.xliff*, where we'll add the localizations for English. Edit the file and make the following changes:

```
<file original="LocalizeMe/Localizable.strings" source-language="en"
datatype="plaintext">
   <header>
     <tool tool-id="com.apple.dt.xcode" tool-name="Xcode" tool-version="7.1.1"
     build-num="7B1005"/>
   </header>
   <body>
<trans-unit id="FLAG_FILE"> <source>FLAG_FILE</source> <note>Name of
the flag</note> <target>flag_usa</target>

     </trans-unit>
     <trans-unit id="LABEL_FIVE">
<source>LABEL_FIVE</source> <note>The number 5</note> <target>Five</
target>

     </trans-unit>
     <trans-unit id="LABEL_FOUR">
<source>LABEL_FOUR</source> <note>The number 4</note> <target>Four</
target>

     </trans-unit>
     <trans-unit id="LABEL_ONE">
<source>LABEL_ONE</source> <note>The number 1</note> <target>One</
target>

     </trans-unit>
     <trans-unit id="LABEL_THREE">
<source>LABEL_THREE</source> <note>The number 3</note>
<target>Three</target>

     </trans-unit>
     <trans-unit id="LABEL_TWO">
```

<source>LABEL_TWO</source> <note>The number 2</note> <target>Two</target>

</trans-unit>
</body>
</file>

Import these changes back into Xcode using **Editor** -> **Import Localizations**.

Note You may find that Xcode is not able to import the development language localizations. A bug has been filed for this problem but, at the time of writing, it remains open. You can temporarily fix the problem by editing the *en.xliff* file, finding every <file> element and adding a target- language attribute to it, with a value of en. There should be five places where you need to make a change. Here's an example of an element that's been modified, with the change highlighted in bold:

<file original="LocalizeMe/Info.plist" source-language="en"
datatype="plaintext"
target-language="en">
Save the file and import it—all should now be well.

Xcode creates a folder called *en.lproj* and adds to it files called *InfoPlist.strings*, *Localizable. strings*, and *Main.strings* that contain the English localization. What you have added is
the reference to the image file for the flag and the text to replace the keys used in the NSLocalizedString() function calls in the code. Now if you run the app with English as your selected language, you'll see the correct English text and the US flag.

There's one more step that you need to take. Switch the simulator or device to a language that's not French or English—say, Spanish—and run the application again. You'll get the same unlocalized

result that you saw when running in English before we added the English localization. That's because when the user's language does not match any of the available localizations, the base localization is used, but we haven't supplied the text strings and flag file to be used in this case. There's a quick solution to this—we can use the English localization to create the base localization. In the Project Navigator, select the file that contains the English variant of *Localizable.strings*, and then in the File Inspector, under Localization, click the **Base** check box to select it. Xcode creates a copy of *Localizable. strings* for the base localization. If you now run the app with Spanish as the active language, it will look just the same as it does in English, which is better than the incomplete version shown in Figure 22-2.

The requirement to provide the flag image file name and the text strings for the base localization arises because we chose not to use localized text as the keys when calling NSLocalizedString(). Had we done something like this, the English text would appear in the user interface for any language for which there is no localization, even if we didn't provide a base localization:

```
labels[0].text = NSLocalizedString("One", comment: "The number 1")
labels[1].text = NSLocalizedString("Two", comment: "The number 2")
labels[2].text = NSLocalizedString("Three", comment: "The number 3")
labels[3].text = NSLocalizedString("Four", comment: "The number 4")
labels[4].text = NSLocalizedString("Five", comment: "The number 5")
let flagFile = NSLocalizedString("flag_usa", comment: "Name of the flag")
```

While this is perfectly legal, the downside is that if you need to change any of the English text strings, you are also changing the key used to look up the strings for all of the other languages, so you will need to manually update all of the localized .strings files so that they use the new key.

Localizing the App Display Name

We want to show you one final piece of localization that is commonly used: localizing the app name that's visible on the home screen and elsewhere. Apple does this for several of the built-in apps, and you might want to do so, as well. The app name used for display is stored in your app's *Info.plist* file, which you'll find in the Project Navigator. Select this file for editing, and you'll see that one of the items it contains, *Bundle name*, is currently set to *$ {PRODUCT_NAME}*. In the syntax used by *Info.plist* files, anything starting with a dollar sign is subject to variable substitution. In this case, it means that when Xcode compiles the app, the value of this item will be replaced with the name of the product in this Xcode project, which is the name of the app itself. This is where we want to do some localization, replacing *${PRODUCT_NAME}* with the localized name for each language. However, as it turns out, this doesn't quite work out as simply as you might expect.

The *Info.plist* file is sort of a special case, and it isn't meant to be localized. Instead, if you want to localize the content of *Info.plist*, you need to make localized versions of a file named *InfoPlist. strings*. Before you can do that, you need to create a Base version of that file. If you followed the steps in the previous section to localize the app, you'll already have English and French versions of this file that are empty. If you don't have these files, you can add one as follows:

1. Select **File** -> **New** -> **File...**, and then in the iOS section, choose **Resource** and then **Strings File**. Press **Next**, name the file *InfoPlist. strings*, assign it to the Supporting Files group in the LocalizeMe project and create it.

2. Select the new file and, in the File Inspector, press **Localize**. In
the dialog box that appears, have the file moved to the English localization and then, back in the File Inspector, check the check box for **French** under Localizations. You should now see copies of this file for both French and English in the

Project Navigator.

We need to add a line to each localized copy of this file to define the display name for the app. In the *Info.plist* file, we were shown the display name associated with a dictionary key called *Bundle name*; however, that's not the real key name! It's merely an Xcode nicety, trying to give us a more friendly and readable name. The real name is CFBundleName, which you can verify by selecting *Info.plist*, right-clicking anywhere in the view, and then selecting **Show Raw Keys/Values**. This shows you the true names of the keys in use. So, select the English localization of *InfoPlist.strings* and either add or modify the following line:

```
"CFBundleName" = "Localize Me";
```

This key may already exist if you followed the localization steps for English, because it's inserted as part of the process of importing an XLIFF file. In fact, another way to localize your app's name is to add the translation to the XLIFF file in the same way as we did for the other texts that we needed to translate—just look for the entry for CFBundleName and add a <trans> element with the translated name. Similarly, select the French localization of the *InfoPlist.strings* file and edit it to give the app a proper French name:

```
"CFBundleName" = "Localisez Moi";
```

Build and run the app, and then go back to the launch screen. And of course, switch the device or simulator you're using to French if it's currently running in English. You should see the localized name just underneath the app's icon, but sometimes it may not appear immediately. iOS seems to cache this information when a new app is added, but it doesn't necessarily change it when an existing app is replaced by a new version—at least not when Xcode is doing the replacing. So, if you're running in French and you don't see the new name—don't worry. Just delete the app from the launch screen, go back to Xcode, and then build and run the app again.

Warning You won't see the localized app name if you are running the application with a custom scheme. The only way to see it is to switch the device or simulator language to French.

Now our application is fully localized for both French and English.

Adding Another Localization

To wrap up, we're going to add another localization to our application. This time, we'll localize it to Swiss French, which is a regional variation of French with language code *fr-CH*. The reason we chose this language is that, at least at the time of writing, it is not one of the languages for which iOS has a specific localization. Nevertheless, you can still localize your app to Swiss French and run it on your iOS device.

The basic principle is the same as before—in fact, now that you have done this once, it should go much faster this time. Start by selecting the project in the Project Navigator, and then select the project itself in the editor, followed by the **Info** tab. In the Localizations section, press **+** to add a new language. You won't see Swiss French in the menu, so scroll down and select **Other**. This opens a submenu with a very large number of languages to choose from—fortunately, they are in alphabetical order. If you scroll down, you will eventually find **French (Switzerland)**, so select it. In the dialog that appears (which looks like Figure 22-5), change the Reference Language for all of the listed files to *French*, so that Xcode uses your existing French translations as the basis for the Swiss French localization, and then click **Finish**. Now if you look at the Project Navigator, you'll see that you have Swiss French versions of the storyboard, localizable strings, and *InfoPlist.strings* files. To demonstrate that this localization is distinct from the French one, open the Swiss French version of *InfoPlist.strings* and change the bundle name to this:

```
"CFBundleName" = "Swiss Localisez Moi";
```

Now build and run the application. Switch to the Settings application and go to Language & Region. As we said earlier, you won't find Swiss French in the list of iPhone Languages. Instead, click **Add Language...** and scroll down (or search) until you find **French (Switzerland)**, and then select it and press **Done**. This will bring up an action sheet in which you will be asked if you prefer Swiss French or your current language. Select **Swiss French** and let iOS reset itself. Go to the home screen and you should now see that our application is called *Swiss Localisez Moi* (in fact, you won't see the whole name, because it's too long, but you get the point ;-)). If you open the application, you'll see that the text is all in French. Unfortunately, the flag is also the French flag, not the Swiss one. By now, you should be able to figure out how to fix this by editing the Swiss localization files. So as an exercise, download a Swiss flag image from the Internet and see if you can make it appear in the Swiss version of the application.

Auf Wiedersehen

If you want to maximize sales of your iOS application, you'll probably want to localize it as much as possible. Fortunately, the iOS localization architecture makes easy work of supporting multiple languages, and even multiple dialects of the same language, within your application. As you saw in this practice, nearly any type of file that you add to your application can be localized.

Even if you don't plan on localizing your application, you should get in the habit of using NSLocalizedString() instead of just using static strings in your code. With Xcode's Code Sense feature, the difference in typing time is negligible. And, should you ever want to translate your application, your life will be much, much easier. Going back late in the project to find all text strings that should be localized is a boring and error-prone process, which you can avoid with a little effort in advance. And on that note, we have now reached the end of our travels together, so it's time for us to say

sayonara, *au revoir*, *auf wiedersehen*, *avtío*, *arrivederci*, *hej då*, , and *adiós*.

The programming language and frameworks are the end result of more than 25 years of evolution. And Apple engineers are feverishly working round the clock, thinking of that next, cool, new thing. The iOS platform has just begun to blossom. There is so much more to come. By making it through this course, you've built yourself a foundation. You have a knowledge of Swift, Cocoa Touch, and the tools that bring these technologies together to create new iPhone, iPod touch, and iPad applications.

Best of luck!