Chapter 22: What's New with User Interface Customization

By Adam Burkepile

With each consecutive version of iOS, Apple has given us greater ways to customize the appearance of our apps. iOS 6 is no exception, adding more customization options for `UISwitch`, `UISlider`, and many more controls. This makes it even easier to make your apps look unique and great!

In this chapter, you're going to take a look at the new changes with user interface customization in iOS 6. Specifically, you're going to examine:

- **New control appearance APIs**. iOS 5 introduced the ability to change the appearance of many controls (without having to subclass them) with the new appearance APIs. iOS 6 has added even more of these – so in this chapter you'll take a look at the new good stuff that got added!
- **Theming your app**. One of the WWDC 2012 demos covered how to arrange your user interface code into a "theme" so it is easy for you to switch to different themes while in development. This isn't a new idea to iOS 6, but is a pretty cool idea and makes your code simpler to update and understand so we're going to cover it here :] And in this chapter, you'll take this even further than the WWDC demo and learn how to allow the user to switch themes dynamically at runtime!
- **Gradient layers**. `CAGradientLayer` provides a neat way to easily apply a gradient to any view in your app – which can be a nice and easy way to make your app look a lot better. This isn't new to iOS 6, but we're going to briefly discuss this here anyway since it's such a useful technique and we haven't covered it in any of our previous tutorials.

To accomplish all this, you're going to work on a basic photo viewing app. The app will start out "plain vanilla" with the default UIKit controls, but then you'll make it legend… (wait for it) …dary through the use of the new iOS 6 customization APIs and dynamic theming!

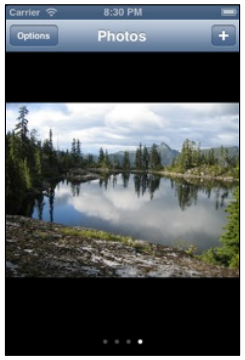
Remember that the focus of this chapter is "what's new in iOS 6", not "everything there is to know about user interface customization". This chapter assumes you already have some basic familiarity with the state of user interface customization in iOS 5. If you are completely new to user interface customization in iOS, you might

want to read the "Beginning/Intermediate UIKit Customization" chapters from *iOS 5 by Tutorials* first.

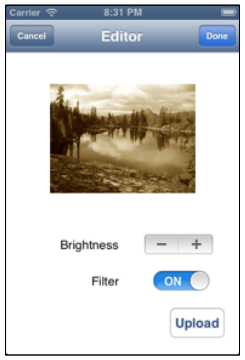
Ready to get customizing? Let's take a stroll and find out what's new with UI customization!

Getting Started

The resources for this chapter include a starter project. Extract the zip, open the project in Xcode, and build and run. You'll see a simple photo editing app that looks like the following:

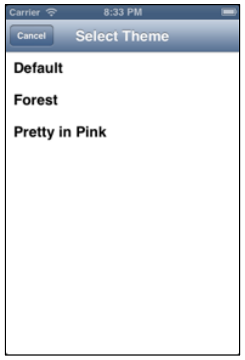


You can swipe left and right to cycle through the photos. You can also tap a photo to adjust its brightnes or to apply a Core Image filter:



You can tap the Upload button to display a progress indicator, but it's just there for show (it doesn't actually do anything).

Finally, if you tap Cancel and then Options, you'll see a screen that allows the user to select a theme to apply:



However, selecting one of the themes doesn't do anything at the moment. That's where you come in! ☺

For the rest of this chapter, you're going to build upon this sample project to take things from "plain vanilla boring" to "themed and delicious."

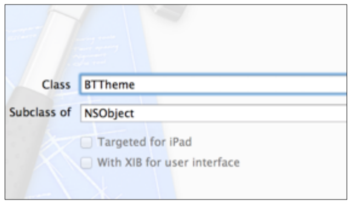
So take a look at the code at this point and make sure you understand how everything fits together. Once you feel comfortable that you understand what's there, read on!

Adding the theme manager

To help give this lackluster app a makeover, you're going to create a "theme manager". Think of this as a crazy (but talented) designer who runs around the app changing the colors and look of each control to make things beautiful.

The advantage of using a theme manager is that it centralizes all of the user interface look and feel logic in one place, which makes it much easier to change and update than the alternative - having it scattered across every view controller in your project!

Let's try this out. Create a new project group and call in **Theme**. Then control-click on the new **Theme** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **BTTheme**, make it a subclass of **NSObject**, and click **Next** and finally **Create**.


Class: BTTheme
Subclass of: NSObject
Targeted for iPad
With XIB for user interface

Open **BTTheme.h** and replace the contents with:

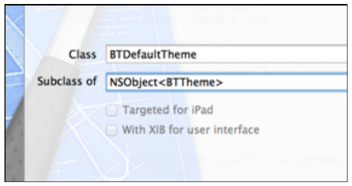
```
@protocol BTTheme

@end

@interface BTThemeManager : NSObject
+ (id<BTTheme>)sharedTheme;
+ (void)setSharedTheme:(id<BTTheme>)inTheme;
+ (void)applyTheme;
@end
```

Here you define two main things:

1. **The theme protocol**. The protocol is called `BTTheme`, and right now is completely empty! You'll be adding new methods to this protocol where each theme can specify how controls should be customized, such as colors or images to apply.

2. **The theme manager**. `BTThemeManager` is the class that will manage the theme and will apply the theme to the controls. Notice that you define methods to get and set the shared theme, both of which deal with an object that conforms to the `BTTheme` protocol.

At this point, it would be helpful to have at least one default theme. Right-click on the **Theme** folder and create a new file. Use it to create a new Objective-C class called **BTDefaultTheme** and have it inherit from **NSObject<BTTheme>**.



Class: BTDefaultTheme
Subclass of: NSObject<BTTheme>

If you haven't seen that <bracket> syntax before, all it means is that this class inherits from `NSObject` while also conforming to the `BTTheme` protocol (there's nothing in that protocol right now, but we have plans for it). Create the class, then add the following `#import` to **BTDefaultThem.h**:

```
#import "BTTheme.h"
```

Now you can jump back to **BTTheme.m** and replace the contents with this:

```
#import "BTTheme.h"
#import "BTDefaultTheme.h"

@implementation BTThemeManager
static id<BTTheme> _theme = nil;

+ (id<BTTheme>)sharedTheme {
    return _theme;
}

+ (void)setSharedTheme:(id<BTTheme>)inTheme {
    _theme = inTheme;
    [self applyTheme];
}

+ (void)applyTheme {
    id<BTTheme> theme = [self sharedTheme];
}
@end
```

The class contains a static variable that holds the theme, and a getter and setter method to access that variable. The last method is the one that will apply the theme to all of the UI controls.

Let's load up the default theme when the app starts. Switch to **BTAppDelegate.m** and add the following line to the beginning of `application:didFinishLaunchingWithOptions:`:

```
[BTThemeManager setSharedTheme:[BTDefaultTheme new]];
```

Don't forget the needed imports at the top:

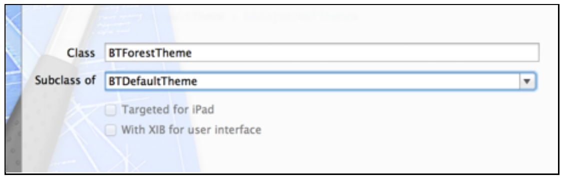
```
#import "BTTheme.h"
#import "BTDefaultTheme.h"
```

It's pretty easy to see what's going on here. When the app starts up, you create a new `BTDefaultTheme` instance and pass that to the shared theme setter, which in turn applies the theme (or at least it will – you'll get to that soon enough).

Build and run to make sure everything works fine, but note that nothing will look or function any differently yet. Patience, my friend!

So now you have a theme manager set up and you are applying a default theme. Let's add another theme so that you have something to switch to.

Control-click on the **Theme** group and once again select **New File.** This will be an **Objective-C class** as well – name it **BTForestTheme**, make it inherit from **BTDefaultTheme**, and click **Next** and finally **Create**.



Class: BTForestTheme
Subclass of: BTDefaultTheme

Now you have two themes! I bet you're feeling powerful already.

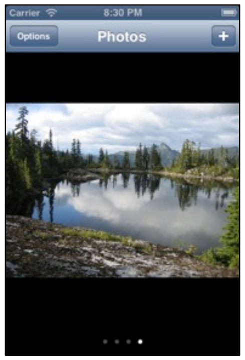
You just need to enable the theme manager to switch between them. To do this, open **BTThemeViewController.m** and add the following imports:

```
#import "BTTheme.h"
#import "BTDefaultTheme.h"
#import "BTForestTheme.h"
```

Then add the following to `tableView:didSelectRowAtIndexPath:` after the first line setting up the `idx` variable:

```
    if ([self.themes[idx] isEqualToString:@"Forest"]) {
        [BTThemeManager setSharedTheme:[BTForestTheme new]];
    }
    else
        [BTThemeManager setSharedTheme:[BTDefaultTheme new]];
```

Give it a build and run. When you select the theme in the theme view controller, the theme manager should change the theme for the whole app.



"But wait a minute," the astute among you might say, "that looks exactly the same as before!"

Right you are – although you now have the scaffolding in place, the themes don't actually do anything yet. Let's customize our first object and change that!

The background

Switch back to **BTTheme.h**. You are going to add a method to the protocol that returns the background for the theme. Add the following to the **BTTheme** protocol (between the `@protocol` and `@end`):

```
- (UIColor*)backgroundColor;
```

Then open **BTDefaultTheme.m** and add the implementation:

```
- (UIColor*)backgroundColor {
    return [UIColor whiteColor];
}
```

In the default theme, you want the background to be a solid white color. Let's specify a different background for the Forest theme. Switch to **BTForestTheme.m** and add:

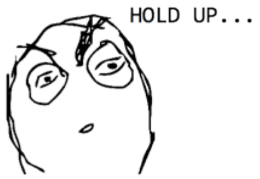
```
- (UIColor*)backgroundColor {
    return [UIColor colorWithPatternImage:
            [UIImage imageNamed:@"bg_forest.png"]];
}
```

This is a little trick to return an image as a color. `UIColors` are nice and easy to work with since all `UIViews` has a `backgroundColor` property – now you can set it to white in the default case, and the bg_forest.png color in the forest theme case.

Now you'll create a method that a view controller can call to set the background color of its views. Open **BTTheme.m** and add the following method:

```
+ (void)customizeView:(UIView *)view
{
    id <BTTheme> theme = [self sharedTheme];
    UIColor *backgroundColor = [theme backgroundColor];
    [view setBackgroundColor:backgroundColor];
}
```

As you're typing this method you might have thought the following:



"Why is there a separate `customizeView` method when there's an existing `applyTheme` method that is supposed to apply the current theme to all the relevant controls?"

Well, later on you'll be using the `applyTheme` method to resolve cases where you want to change the look of all controls of a certain type, globally. For example, you might want to change all the navigation bars to have a different background.

But in this case, a lot of standard UI controls are sub-classes of `UVIew` – for instance, `UIButton`, `UILabel`, etc. And if you were to globally set the background for all views, then you'd discover that controls you might not have expected will also have their backgroudn changed! And this does not maker our designer happy. ☺

So, instead you add a special method here that will change the background for only specific views. This gives you control over which views are customized and which aren't!

Don't forget to add the prototype for the above method to **BTTheme.h**:

```
+ (void)customizeView:(UIView *)view;
```

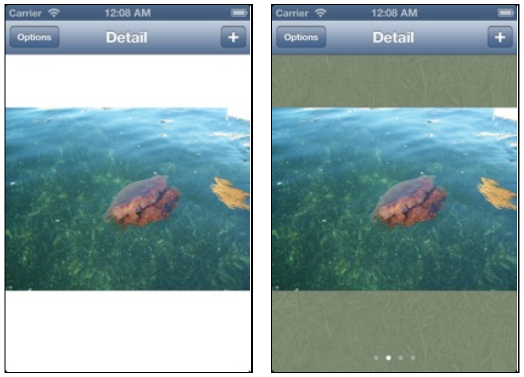
Now in **BTThemeViewController.m**, **BTPhotosViewController.m**, **BTPhotoViewController.m**, and **BTEditorViewController.m**, add the following line to the end of `viewWillAppear:`:

```
[BTThemeManager customizeView:self.view];
```

You will need to add the following import at the top of **all** of the above files, *except* for BTThemeViewController.m:

```
#import "BTTheme.h"
```

Build and run. Try switching to the Forest theme now.



BOOM - the results of all your efforts so far – a leafy background!

Although that is a fine looking leafy background, you might not be super impressed since all you did was change the background. But don't worry - now you have a solid framework that will make the rest of the customization easy.

You can see that your theme manager is working, storing the shared theme when you change it, and accessing and applying the properties from the theme. Now for the fun part - customizing the rest of the controls!

Navigation bar, meet proxy

In iOS 5 Apple introduced some APIs to allow you to change the appearance of a `UINavigationBar`, such as changing its background and title text attributes. In iOS 6 Apple has made this even better – navigation bars now have shadows that are fully customizable!

Before you get to changing the shadow on the navigation bar, let's start with changing the background, which will be a review of iOS 5 customization. Switch to **BTTheme.h** and add new method definitions to the protocol:

```
- (UIImage*)imageForNavigationBar;
- (UIImage*)imageForNavigationBarLandscape;

- (NSDictionary*)navBarTextDictionary;
```

Here you have getters for the `UINavigationBar` images for portrait and landscape, and a dictionary that will set the attributes of the title text (font, color, etc.) for the bar.

Add the following to **BTDefaultTheme.m**:

```
- (UIImage*)imageForNavigationBar{return nil;}
- (UIImage*)imageForNavigationBarLandscape{return nil;}
- (NSDictionary*)navBarTextDictionary { return nil; }
```

When you set the properties to `nil`, you just get the default appearances.

Now add the following to **BTForestTheme.m**:

```
- (UIImage*)imageForNavigationBar {
    return [[UIImage imageNamed:@"nav_forest_portrait.png"]
             resizableImageWithCapInsets:UIEdgeInsetsMake(
                                     0.0, 100.0, 0.0, 100.0)];
}
```

```
- (UIImage*)imageForNavigationBarLandscape{
    return [[UIImage imageNamed:@"nav_forest_landscape.png"]
            resizableImageWithCapInsets:UIEdgeInsetsMake(
                                    0.0, 100.0, 0.0, 100.0)];
}
- (NSDictionary*)navBarTextDictionary { return @{
  UITextAttributeFont:[UIFont fontWithName:@"Optima" size:24.0],
  UITextAttributeTextColor:
    [UIColor colorWithRed:0.910 green:0.914 blue:0.824
    alpha:1.000],
  UITextAttributeTextShadowColor:
    [UIColor colorWithRed:0.224 green:0.173 blue:0.114
    alpha:1.000],
  UITextAttributeTextShadowOffset:
    [NSValue valueWithUIOffset:UIOffsetMake(0, -1)]
    };
}
```

The above code returns the images to use for the navigation bar in both portrait and landscape, and the properties to use for the navigation bar text.

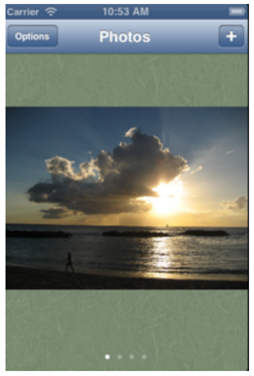
Now switch to **BTTheme.m** and add the following to `applyTheme`:

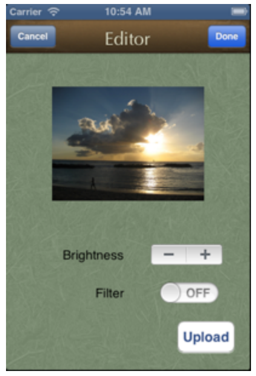
```
UINavigationBar* NavBarAppearence =
  [UINavigationBar appearance];
[NavBarAppearence setBackgroundImage:[theme
                                    imageForNavigationBar]
                  forBarMetrics:UIBarMetricsDefault];
[NavBarAppearence setBackgroundImage:[theme
                                imageForNavigationBarLandscape]
                  forBarMetrics:UIBarMetricsLandscapePhone];
[NavBarAppearence setTitleTextAttributes:[theme
                                       navBarTextDictionary]];
```

And here you have your first use of the appearance proxy – remember this is the way to change the appearance of *all* controls of a particular type across the app. The above code gets the appearance proxy for the `UINavigationBar` class, and then sets the default appearance for all navigation bars to fit the current theme.

Build and run. Switch to the Forest theme now. Hmm, it didn't seem to work!

But hold on a second. When you tap on a photo and open up the Editor view, you can see that the navigation bar has changed!





Furthermore, when you close the Editor, the navigation bar in the Photos view has changed too! What gives?

This is something you have to be aware of when you use appearance proxies to customize appearances. When you set a property through the appearance proxy, it only applies to *new* objects that are added to the window. It will not automatically apply itself to objects already in the displayed window. You need to update those yourself.

To do this, you're going to use an `NSNotification` to send a message to the app delegate when you switch themes. Add the following to **BTTheme.m** in the `BTThemeManager` class:

```
+ (void)customizeNavigationBar:(UINavigationBar *)navigationBar {
    id <BTTheme> theme = [self sharedTheme];
    [navigationBar setBackgroundImage:[theme
                                      imageForNavigationBar]
                   forBarMetrics:UIBarMetricsDefault];
    [navigationBar setBackgroundImage:[theme
                                imageForNavigationBarLandscape]
                   forBarMetrics:UIBarMetricsLandscapePhone];
    [navigationBar setTitleTextAttributes:[theme
                                        navBarTextDictionary]];
}
```

The above method customizes the look of a specific navigation bar instance, rather than all the navigation bars.

Even though it might make that little guy sad, it's all for the best. When a theme changes, you'll send out a notification which the app delegate will be listening for. When the notification occurs, the the app delecate can call this method to update the previously created navigation bar appropriately.

Also add the method declaration/prototype in **BTTheme.h**:

```
+ (void)customizeNavigationBar:(UINavigationBar *)navigationBar;
```

Next, open **Appearence-Prefix.pch** (it's in the Supporting Files group). This is the precompiled header that gets added to each file at compile time. If you have something that needs to be in a lot of the files in the project, this is a good place to add it.

Put this line at the end of the file (after the `#endif`):

```
#define kThemeChange @"NotificationThemeChanged"
```

Now add the following to **BTAppDelegate.m** at the end of `application:didFinishLaunchingWithOptions:` (but before the final `return`):

```
[[NSNotificationCenter defaultCenter]
             addObserverForName:kThemeChange
                         object:nil
                          queue:[NSOperationQueue mainQueue]
                     usingBlock:^(NSNotification *note) {
              UINavigationController* navController =
   (UINavigationController*)[[[UIApplication sharedApplication]
                                 keyWindow] rootViewController];

[BTThemeManager customizeNavigationBar:
                                  navController.navigationBar];
}];
```

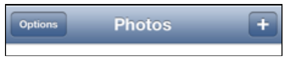
This sets up the "listener" part of the notification and calls the `customizeNavigationBar` method you just set up on the main app `UINavigationBar` when a notification about theme changes is received. This way, you can ensure that when your theme changes, the navigation bar theme is also explicitly changed.

Now, add the following line to `applyTheme` in **BTTheme.m**:

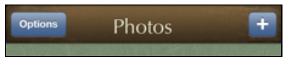
```
[[NSNotificationCenter defaultCenter]
                                postNotification:[NSNotification
                            notificationWithName:kThemeChange
                                          object:nil]];
```

This is the part that sends out the notification to anyone listening for it. So, whenever you change themes, you call the `applyTheme` method, which sends the `kThemeChange` notification, which the AppDelegate then picks up and calls the `customizeNavigationBar` method to set the navigation bar. Pretty cool, eh?

Build and run, and everything should change as expected:



Default Theme



Forest Theme

Whoohoo – now it's starting to come together!

Next, let's change the shadow below the navigation bar. It's a subtle thing added in iOS 6, but it's a nice touch that you shouldn't neglect. Add the following code to the appropriate files:

To the `BTTheme` protocol in **BTTheme.h**:

```
- (UIImage*)imageForNavigationBarShadow;
```

To **BTDefaultTheme.m**:

```
- (UIImage*)imageForNavigationBarShadow{return nil;}
```

To **BTForestTheme.m**:

```
-  (UIImage*)imageForNavigationBarShadow{
    return [UIImage imageNamed:@"topShadow_forest.png"];
}
```

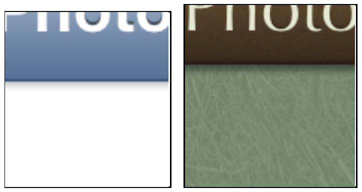
To `applyTheme` in **BTTheme.m**:

```
[NavBarAppearence setShadowImage:[theme
                                  imageForNavigationBarShadow]];
```

And to `customizeNavigationBar` in **BTTheme.m**:

```
  [navigationBar setShadowImage:[theme
                                   imageForNavigationBarShadow]];
```

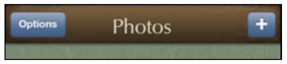
Build and run.



Like I said, subtle. But these little details make an impression!

Bar button items

So you got the navigation bar customized, but it looks pretty weird with the buttons not matching the Forest theme. So let's look at customizing those next.



iOS 5 gave us an easy way to customize bar buttons, and in iOS 6 Apple has added customization of the different button styles. You'll get to see examples of those in the Editor view. Add the following code to the `BTTheme` protocol in **BTTheme.h**:

```
 - (UIImage*)imageForBarButtonNormal;
 - (UIImage*)imageForBarButtonHighlighted;
 - (UIImage*)imageForBarButtonNormalLandscape;
 - (UIImage*)imageForBarButtonHighlightedLandscape;
 - (UIImage*)imageForBarButtonDoneNormal;
 - (UIImage*)imageForBarButtonDoneHighlighted;
 - (UIImage*)imageForBarButtonDoneNormalLandscape;
 - (UIImage*)imageForBarButtonDoneHighlightedLandscape;
 - (NSDictionary*)barButtonTextDictionary;
```

As you can see, you have a few more properties this time. You have methods to return images for a standard button's normal and highlighted states, a done button's normal and highlighted states, and the corresponding landscape versions for each of those buttons, for a total of 8 different images that need to be provided. In addition, there's a method that will set the text attributes for a bar button item.

Add to **BTDefaultTheme.m**:

```
 - (UIImage*)imageForBarButtonNormal { return nil; }
 - (UIImage*)imageForBarButtonHighlighted { return nil; }
 - (UIImage*)imageForBarButtonNormalLandscape { return nil; }
 - (UIImage*)imageForBarButtonHighlightedLandscape {return nil;}
 - (UIImage*)imageForBarButtonDoneNormal { return nil; }
 - (UIImage*)imageForBarButtonDoneHighlighted { return nil; }
 - (UIImage*)imageForBarButtonDoneNormalLandscape { return nil; }
 - (UIImage*)imageForBarButtonDoneHighlightedLandscape
                                                    { return nil; }
 - (NSDictionary*)barButtonTextDictionary  {
   return @{
     UITextAttributeFont:[UIFont fontWithName:@"Helvetica-Bold"
                                        size:12.0f],
     UITextAttributeTextColor:[UIColor whiteColor],
     UITextAttributeTextShadowColor:[UIColor blackColor],
     UITextAttributeTextShadowOffset:[NSValue
       valueWithUIOffset:UIOffsetMake(0, -1)]
   };
 }
```

And to **BTForestTheme.m**:

```
 - (UIImage*)imageForBarButtonNormal {
         return [[UIImage imageNamed:@"barbutton_forest_uns.png"]
                   resizableImageWithCapInsets:UIEdgeInsetsMake(
                                        0.0, 5.0, 0.0, 5.0)]; }
 - (UIImage*)imageForBarButtonHighlighted {
         return [[UIImage imageNamed:@"barbutton_forest_sel.png"]
                   resizableImageWithCapInsets:UIEdgeInsetsMake(
                                        0.0, 5.0, 0.0, 5.0)]; }
 - (UIImage*)imageForBarButtonDoneNormal {
    return [[UIImage imageNamed:@"barbutton_forest_done_uns.png"]
                   resizableImageWithCapInsets:UIEdgeInsetsMake(
                                        0.0, 5.0, 0.0, 5.0)]; }
 - (UIImage*)imageForBarButtonDoneHighlighted {
    return [[UIImage imageNamed:@"barbutton_forest_done_sel.png"]
                   resizableImageWithCapInsets:UIEdgeInsetsMake(
                                        0.0, 5.0, 0.0, 5.0)]; }
 - (UIImage*)imageForBarButtonNormalLandscape {
          return [[UIImage imageNamed:
                            @"barbutton_forest_landscape_uns.png"]
                    resizableImageWithCapInsets:UIEdgeInsetsMake(
                                        0.0, 5.0, 0.0, 5.0)]; }
 - (UIImage*)imageForBarButtonHighlightedLandscape {
         return [[UIImage imageNamed:
                          @"barbutton_forest_landscape_sel.png"]
                   resizableImageWithCapInsets:UIEdgeInsetsMake(
                                         0.0, 5.0, 0.0, 5.0)]; }
- (UIImage*)imageForBarButtonDoneNormalLandscape {
         return [[UIImage imageNamed:
                     @"barbutton_forest_done_landscape_uns.png"]
                   resizableImageWithCapInsets:UIEdgeInsetsMake(
                                         0.0, 5.0, 0.0, 5.0)]; }
- (UIImage*)imageForBarButtonDoneHighlightedLandscape {
         return [[UIImage imageNamed:
                     @"barbutton_forest_done_landscape_sel.png"]
                   resizableImageWithCapInsets:UIEdgeInsetsMake(
                                         0.0, 5.0, 0.0, 5.0)]; }
- (NSDictionary*)barButtonTextDictionary  {
return @{
UITextAttributeFont:[UIFont fontWithName:@"Optima" size:18.0],
UITextAttributeTextColor:[UIColor colorWithRed:0.965 green:0.976
                                  blue:0.875 alpha:1.000],
UITextAttributeTextShadowColor:[UIColor colorWithRed:0.224
                            green:0.173 blue:0.114 alpha:1.000],
UITextAttributeTextShadowOffset:[NSValue
                           valueWithUIOffset:UIOffsetMake(0, 1)]
    };
}
```

This returns the images for the buttons in their normal and highlighted states, for both normal and landscape. Note that there is a separate case for done – that is the part that's new in iOS 6. You can now have separate images for different bar button styles (like the done style).

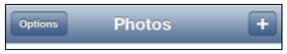
And finally, add this to the bottom of `applyTheme` in **BTTheme.m**:

```
UIBarButtonItem* barButton = [UIBarButtonItem appearance];

[barButton setBackgroundImage:[theme imageForBarButtonNormal]
                     forState:UIControlStateNormal
                   barMetrics:UIBarMetricsDefault];
[barButton setBackgroundImage:[theme
                      imageForBarButtonHighlighted]
                     forState:UIControlStateHighlighted
                   barMetrics:UIBarMetricsDefault];
[barButton setBackgroundImage:[theme
                      imageForBarButtonNormalLandscape]
                     forState:UIControlStateNormal
                   barMetrics:UIBarMetricsLandscapePhone];
```

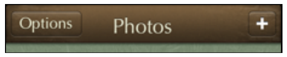
```
[barButton setBackgroundImage:[theme
                          imageForBarButtonHighlightedLandscape]
                     forState:UIControlStateHighlighted
                   barMetrics:UIBarMetricsLandscapePhone];

 [barButton setBackgroundImage:[theme
                   imageForBarButtonDoneNormal]
                   forState:UIControlStateNormal
                   style:UIBarButtonItemStyleDone
                   barMetrics:UIBarMetricsDefault];
 [barButton setBackgroundImage:[theme
                   imageForBarButtonDoneHighlighted]
                   forState:UIControlStateHighlighted
                   style:UIBarButtonItemStyleDone
                   barMetrics:UIBarMetricsDefault];
 [barButton setBackgroundImage:[theme
                    imageForBarButtonDoneNormalLandscape]
                   forState:UIControlStateNormal
                   style:UIBarButtonItemStyleDone
                   barMetrics:UIBarMetricsLandscapePhone];
 [barButton setBackgroundImage:[theme
                   imageForBarButtonDoneHighlightedLandscape]
                   forState:UIControlStateHighlighted
                   style:UIBarButtonItemStyleDone
                   barMetrics:UIBarMetricsLandscapePhone];
 [barButton setTitleTextAttributes:[theme
                   barButtonTextDictionary]
                   forState:UIControlStateNormal];

 [barButton setTitleTextAttributes:[theme
                                        barButtonTextDictionary]
                          forState:UIControlStateNormal];
```

Here you'll see that there is a new `setBackgroundImage:forState:style:barMetrics` method, that allows you to pass in the style of the button to apply the background image (the done style in this case).

Give it a build and run. Ah, that's looks much better now!



Default Theme



Forest Theme

Paging custom control, do you copy?

Now let's look at the last remaining element to customize on the Photos View Controller, the page control at the bottom.

In iOS 5 you couldn't (easily) customize this, but now you can! iOS 6 provides a `tintColor` and `currentTintColor` property `UIPageControl`.

In the default theme, you can't even see the control because it's white-on-white, so that definitely needs to change. The Forest theme control could also benefit from a color change so that it stands out a bit more.

Add the following code to `BTTheme` protocol in **BTTheme.h**:

```
- (UIColor*)pageTintColor;
- (UIColor*)pageCurrentTintColor;
```

To **BTDefaultTheme.m**:

```
- (UIColor*)pageTintColor {
      return [UIColor lightGrayColor];
}
- (UIColor*)pageCurrentTintColor {
      return [UIColor blackColor];
}
```

To **BTForestTheme.m**:

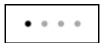
```
- (UIColor*)pageTintColor {
                 return [UIColor colorWithRed:0.973 green:0.984
                                        blue:0.875 alpha:1.000];
}
- (UIColor*)pageCurrentTintColor {
                 return [UIColor colorWithRed:0.063 green:0.169
                                        blue:0.071 alpha:1.000];
}
```

And to `applyTheme` in **BTTheme.m**:

```
UIPageControl* pageAppearence = [UIPageControl appearance];
 [pageAppearence setCurrentPageIndicatorTintColor:
                              [theme pageCurrentTintColor]];
```

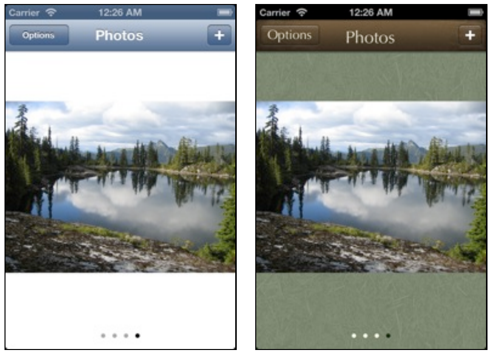
```
  [pageAppearence setPageIndicatorTintColor:
                                  [theme pageTintColor]];
```

Build and run. You should now see the page control with the default theme:

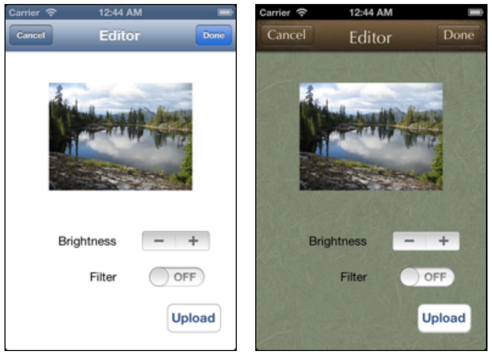


And with the Forest theme:

And with that, you've customized all of the controls on the Photos View Controller. Take a bow!



Looks pretty nice, huh? But this chapter's not over, which means it's time to move on to the Editor View Controller.



As you can see, all the customizations that you've already done are also present in this view (since the appearance proxy changes the look for the relevant controls globally), but some of the controls on this view still need customizing. Let's take them one-by-one.

Stepping out

Begin with the `UIStepper` control.

This control was introduced in iOS 5, but it wasn't very customizable (by "very" I mean "at all"). With iOS 6, you now have access to the standard tint color, as well as the ability to set custom images for the plus and minus signs, the background for the control, and the middle divider.

Customize the `UIStepper` by adding the following code, first to the `BTTheme` protocol in **BTTheme.h**:

```
- (UIImage*)imageForStepperUnselected;
- (UIImage*)imageForStepperSelected;
- (UIImage*)imageForStepperDecrement;
- (UIImage*)imageForStepperIncrement;
- (UIImage*)imageForStepperDividerUnselected;
- (UIImage*)imageForStepperDividerSelected;
```

To **BTDefaultTheme.m**:

```
- (UIImage*)imageForStepperUnselected{return nil;}
- (UIImage*)imageForStepperSelected{return nil;}
- (UIImage*)imageForStepperDecrement{return nil;}
- (UIImage*)imageForStepperIncrement{return nil;}
- (UIImage*)imageForStepperDividerUnselected{return nil;}
- (UIImage*)imageForStepperDividerSelected{return nil;}
```

To **BTForestTheme.m**:

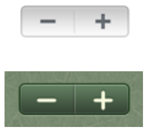
```
- (UIImage*)imageForStepperUnselected{
  return [UIImage imageNamed:@"stepper_forest_bg_uns.png"];
}
- (UIImage*)imageForStepperSelected{
  return [UIImage imageNamed:@"stepper_forest_bg_sel.png"];
}
- (UIImage*)imageForStepperDecrement{
  return [UIImage imageNamed:@"stepper_forest_decrement.png"];
}
- (UIImage*)imageForStepperIncrement{
  return [UIImage imageNamed:@"stepper_forest_increment.png"];
}
- (UIImage*)imageForStepperDividerUnselected{
  return [UIImage imageNamed:@"stepper_forest_divider_uns.png"];
}
- (UIImage*)imageForStepperDividerSelected{
  return [UIImage imageNamed:@"stepper_forest_divider_sel.png"];
}
```

And to `applyTheme` in **BTTheme.m**:

```
UIStepper* stepperAppearence = [UIStepper appearance];
[stepperAppearence setBackgroundImage:
                 [theme imageForStepperUnselected]
               forState:UIControlStateNormal];
[stepperAppearence setBackgroundImage:
                 [theme imageForStepperSelected]
               forState:UIControlStateHighlighted];
[stepperAppearence setDividerImage:
                 [theme imageForStepperDividerUnselected]
               forLeftSegmentState:UIControlStateNormal
               rightSegmentState:UIControlStateNormal];
[stepperAppearence setDividerImage:
                 [theme imageForStepperDividerSelected]
               forLeftSegmentState:UIControlStateSelected
               rightSegmentState:UIControlStateNormal];
[stepperAppearence setDividerImage:
```

```
                 [theme imageForStepperDividerSelected]
               forLeftSegmentState:UIControlStateNormal
               rightSegmentState:UIControlStateSelected];
[stepperAppearence setDividerImage:
                 [theme imageForStepperDividerSelected]
               forLeftSegmentState:UIControlStateSelected
               rightSegmentState:UIControlStateSelected];
[stepperAppearence setIncrementImage:
                 [theme imageForStepperIncrement]
               forState:UIControlStateNormal];
[stepperAppearence setDecrementImage:
                 [theme imageForStepperDecrement]
               forState:UIControlStateNormal];
```

Build and run. You should now see your customized stepper control:



Switching it up

The switch is an interesting control that has been with us since its introduction in iOS 2, but it didn't have much customizability until iOS 5. Even then, it was just the tint color for one side of the control.

With iOS 6, you have much more robust control over the appearance of the switch, like being able to set the images and separate tint colors for each side, and to set the thumb image.

Add the following code, first to the `BTTheme` protocol in **BTTheme.h**:

```
- (UIColor*)switchOnTintColor;
- (UIColor*)switchThumbTintColor;
- (UIImage*)imageForSwitchOn;
- (UIImage*)imageForSwitchOff;
```

To **BTDefaultTheme.m**:

```
- (UIColor*)switchOnTintColor { return nil; }
- (UIColor*)switchThumbTintColor { return nil; }
- (UIImage*)imageForSwitchOn{return nil;}
- (UIImage*)imageForSwitchOff{return nil;}
```

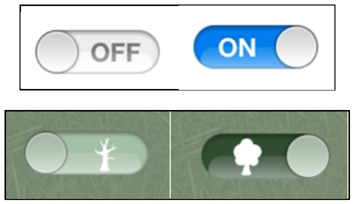
To **BTForestTheme.m**:

```
- (UIColor*)switchOnTintColor {
    return [UIColor colorWithRed:0.192 green:0.298
                           blue:0.200 alpha:1.000];
}
- (UIColor*)switchThumbTintColor {
    return [UIColor colorWithRed:0.643 green:0.749
                           blue:0.651 alpha:1.000];
}
- (UIImage*)imageForSwitchOn{
    return [UIImage imageNamed:@"tree_on.png"];
}
- (UIImage*)imageForSwitchOff{
    return [UIImage imageNamed:@"tree_off.png"];
}
```

And to `applyTheme` in **BTTheme.m**:

```
UISwitch* switchAppearence = [UISwitch appearance];
 [switchAppearence setTintColor:[theme switchThumbTintColor]];
 [switchAppearence setOnTintColor:[theme switchOnTintColor]];
 [switchAppearence setOnImage:[theme imageForSwitchOn]];
 [switchAppearence setOffImage:[theme imageForSwitchOff]];
```

Build and run. Go to the Editor screen for both themes. It's a pretty simple and straightforward customization.



Are you more of a winter tree person, or a summer tree person? That might say a lot about your personality. ☺

Progressing to the progress view

The progress view is a control that has had good customization options since iOS 5. You can customize the track color and completed track color. Nothing is new in iOS

6, but we haven't covered this before so we thought we'd show it off real quick here.

Add the following to the `BTTheme` protocol in **BTTheme.h**:

```
- (UIColor*)progressBarTintColor;
- (UIColor*)progressBarTrackTintColor;
```

To **BTDefaultTheme.m**:

```
- (UIColor*)progressBarTintColor { return nil; }
- (UIColor*)progressBarTrackTintColor { return nil; }
```

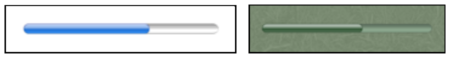
To **BTForestTheme.m**:

```
- (UIColor*)progressBarTintColor {
    return [UIColor colorWithRed:0.200 green:0.345
                            blue:0.212 alpha:1.000];
}
- (UIColor*)progressBarTrackTintColor {
    return [UIColor colorWithRed:0.541 green:0.647
                            blue:0.549 alpha:1.000];
}
```

And to `applyTheme` in **BTTheme.m**:

```
    UIProgressView* progressAppearence = [UIProgressView
appearance];
    [progressAppearence setProgressTintColor:[theme
progressBarTintColor]];
    [progressAppearence setTrackTintColor:[theme
progressBarTrackTintColor]];
```

Build and run. You should see the new progress view now in the Forest theme.



Labels, which can be cranky

The standard label control can be difficult to work with at times. You could always set basic color and font properties, but more complex attributed strings were not available until iOS 6, with a slew of third-party libraries trying to fill the gap.

For this chapter, you're just going to focus on the font and color properties, but this book has a ton of more information about cool things you can do with atributed strings! If you want to learn more, check out Chapter 15, "Attributed Strings."

Let's customize the labels on the Editor view controller to match the themes. First, add the following to the `BTTheme` protocol in **BTTheme.h**:

```
- (NSDictionary*)labelTextDictionary;
```

To **BTDefaultTheme.m**:

```
- (NSDictionary*)labelTextDictionary { return nil; }
```

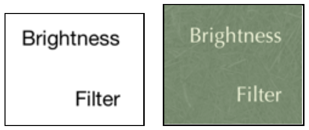
To **BTForestTheme.m**:

```
- (NSDictionary*)labelTextDictionary  {
  return @{
    UITextAttributeFont:[UIFont fontWithName:@"Optima"
                                        size:18.0],
    UITextAttributeTextColor:[UIColor colorWithRed:0.965
                          green:0.976 blue:0.875 alpha:1.000],
    UITextAttributeTextShadowColor:[UIColor colorWithRed:0.212
                          green:0.263 blue:0.208 alpha:1.000],
    UITextAttributeTextShadowOffset:
                  [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
  };
}
```

And to `applyTheme` in **BTTheme.m**:

```
UILabel* labelAppearence = [UILabel appearance];
[labelAppearence setTextColor:
         [theme labelTextDictionary][UITextAttributeTextColor]];
[labelAppearence setFont:
         [theme labelTextDictionary][UITextAttributeFont]];
```

Build and run again. You should now see the new font and color on the Editor screen.



However, you will also start noticing some weird issues at this point – the table view for displaying the themes will be formatted differently, and the button text on the bar button items will show different fonts at times.

These are both known issues of globally-customizing labels, since labels are used as internal building blocks for other controls, such as table cell views. So it's definitely something to watch out for.

Since you already know how to modify individual controls (the same way view backgrounds were set, and the same way buttons will be customized in the next section), I'll leave it as an exercise for you to modify the implementation to customize just the labels you want.

Buttons, one at a time

Ever since the beginning of iOS, it's been easy to customize buttons, since it's as simple as setting the background image of a button. But iOS 5 made it easy to globally update buttons with the `UIAppearance` proxy. In this section, you'll update the button with a resizable image.

You'll supply both in your themes. First add the following to the `BTTheme` protocol in **BTTheme.h**:

```
- (NSDictionary*)buttonTextDictionary;
- (UIImage*)imageForButtonNormal;
- (UIImage*)imageForButtonHighlighted;
```

Then to the `BTThemeManager` interface in **BTTheme.h**:

```
+ (void)customizeButton:(UIButton*)button;
```

Buttons are similar to views, in that they are so ubiquitous, you don't want to try customizing them globally – you just wouldn't like the result. ☺ So here you add a method that allows you to customize the look of individual buttons.

Add this to **BTDefaultTheme.m**:

```
- (NSDictionary*)buttonTextDictionary { return nil; }
- (UIImage*)imageForButtonNormal { return nil; }
- (UIImage*)imageForButtonHighlighted { return nil; }
```

To **BTForestTheme.m**:

```
- (NSDictionary*)buttonTextDictionary  { return @{
UITextAttributeFont:[UIFont fontWithName:@"Optima" size:15.0],
UITextAttributeTextColor:
                    [UIColor colorWithRed:0.965 green:0.976
                                    blue:0.875 alpha:1.000],
UITextAttributeTextShadowColor:
```

```
                     [UIColor colorWithRed:0.212 green:0.263
                                     blue:0.208 alpha:1.000],
UITextAttributeTextShadowOffset:
    [NSValue valueWithUIOffset:UIOffsetMake(0, -1)]
    };
}
- (UIImage*)imageForButtonNormal {
    return [[UIImage imageNamed:@"button_forest_uns.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 8.0, 0.0, 8.0)];
}
- (UIImage*)imageForButtonHighlighted {
    return [[UIImage imageNamed:@"button_forest_sel.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 8.0, 0.0, 8.0)];
}
```

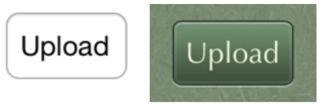
And to **BTTheme.m**:

```
+ (void)customizeButton:(UIButton*)button {
    id <BTTheme> theme = [self sharedTheme];

    [button setTitleColor:
     [theme buttonTextDictionary][UITextAttributeTextColor]
                   forState:UIControlStateNormal];
    [[button titleLabel] setFont:
     [theme buttonTextDictionary][UITextAttributeFont]];
    [button setBackgroundImage:
     [theme imageForButtonNormal]?
     [theme imageForButtonNormal] : nil
                 forState:UIControlStateNormal];
    [button setBackgroundImage:
     [theme imageForButtonHighlighted]?
     [theme imageForButtonHighlighted]:nil
                 forState:UIControlStateHighlighted];
}
```

Because you only want to apply the customization to the Upload button on the Editor view controller, add the following line in `viewDidLoad:` in **BTEditorViewController.m**:

```
[BTThemeManager customizeButton:ibUploadButton];
```

Build and run, and check out the results:



And just like that, you should now have a beautiful Upload button in the Forest theme.

Gradient layers in table view cells

You're now going to turn your attention to the final part of the app that needs to be themed, the theme selection view controller itself. And really, all that's left to be customized are the table view cells, since you've already taken care of the navigation bar and the buttons.

This will be a little different than the other customizations you've done in this chapter, because you are going to use a `CAGradientLayer` to create a nice-looking background for the table view cells. You'll specify the colors you need, the text properties, and a gradient layer class that uses the colors to describe the gradient that should be used with the cell.

First add the following to the `BTTheme` protocol in **BTTheme.h**:

```
- (UIColor*)upperGradient;
- (UIColor*)lowerGradient;
- (UIColor*)seperatorColor;
- (Class)gradientLayer;
- (NSDictionary*)tableViewCellTextDictionary;
```

Then to the `BTThemeManager` interface in **BTTheme.h**:

```
+ (void)customizeTableViewCell:(UITableViewCell*)tableViewCell;
```

To **BTDefaultTheme.m**:

```
- (UIColor*)upperGradient{
    return [UIColor whiteColor];
}
- (UIColor*)lowerGradient {
    return [UIColor whiteColor];
}
- (UIColor*)seperatorColor {
    return [UIColor blackColor];
}
- (Class)gradientLayer {
```

```
    return [BTGradientLayer class];
}
- (NSDictionary*)tableViewCellTextDictionary {
     return @{
          UITextAttributeFont:[UIFont boldSystemFontOfSize:20.0]
     };
}
```

To **BTForestTheme.m**:

```
- (UIColor*)upperGradient{
    return [UIColor colorWithRed:0.976 green:0.976
                            blue:0.937 alpha:1.000];
}
- (UIColor*)lowerGradient {
    return [UIColor colorWithRed:0.969 green:0.976
                            blue:0.878 alpha:1.000];
}
- (UIColor*)seperatorColor {
    return [UIColor colorWithRed:0.753 green:0.749
                            blue:0.698 alpha:1.000];
}
- (NSDictionary*)tableViewCellTextDictionary { return @{
UITextAttributeFont:[UIFont fontWithName:@"Optima" size:24.0],
UITextAttributeTextColor:
                      [UIColor colorWithRed:0.169 green:0.169
                                      blue:0.153 alpha:1.000],
UITextAttributeTextShadowColor:
                     [UIColor colorWithWhite:1.000 alpha:1.000],
UITextAttributeTextShadowOffset:
                 [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
    };
}
```

And to **BTTheme.m**:

```
+ (void)customizeTableViewCell:(UITableViewCell*)tableViewCell {
    id <BTTheme> theme = [self sharedTheme];

    [[tableViewCell textLabel] setTextColor:
 [theme tableViewCellTextDictionary][UITextAttributeTextColor]];
    [[tableViewCell textLabel] setFont:
 [theme tableViewCellTextDictionary][UITextAttributeFont]];
}
```

Finally, add the following to `tableView:cellForRowAtIndexPath:` (right after the cell initialization) in **BTThemeViewController.m**:

```
[BTThemeManager customizeTableViewCell:cell];
```

So you have set up the font properties you need and are applying them to each cell when the cell gets retrieved for the table view. You have also defined the colors you are going to be using for the gradient background,

But where's the gradient background? Time to create it! Add the following code to **BTDefaultTheme.h**:

```
#import <QuartzCore/QuartzCore.h>

@interface BTGradientLayer : CAGradientLayer
@end
```

And to **BTDefaultTheme.m** (after the final @end in the file):

```
@implementation BTGradientLayer
- (id)init{
    if(self = [super init]) {
        UIColor *colorOne = [[BTThemeManager sharedTheme]
                                                upperGradient];
        UIColor *colorTwo = [[BTThemeManager sharedTheme]
                                                lowerGradient];
        UIColor *colorThree = [[BTThemeManager sharedTheme]
                                               seperatorColor];

        NSArray *colors =  [NSArray arrayWithObjects:
                                (id)colorOne.CGColor,
                                    colorTwo.CGColor,
                                    colorThree.CGColor, nil];

        self.colors = colors;

        NSNumber *stopOne = [NSNumber numberWithFloat:0.0];
        NSNumber *stopTwo = [NSNumber numberWithFloat:0.98];
        NSNumber *stopThree = [NSNumber numberWithFloat:1.0];

        NSArray *locations = [NSArray arrayWithObjects:
                                 stopOne,
                                 stopTwo,
                                 stopThree, nil];

        self.locations = locations;
```

```
        self.startPoint = CGPointMake(0.5, 0.0);
        self.endPoint = CGPointMake(0.5, 1.0);
    }

    return self;
}
@end
```

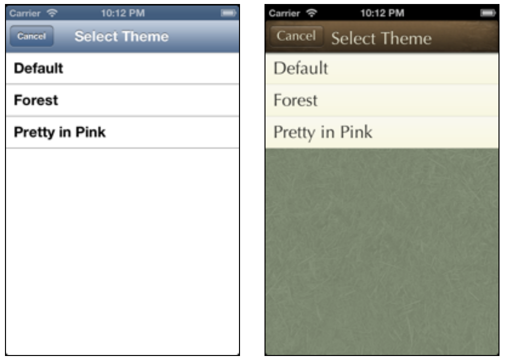
It's not too hard to see what a `CAGradientLayer` does or how it works. You set up an array of colors, which you are able to pull from your shared theme, and you also set up an array that specifies the positions where each color falls.

Then to implement your custom gradient layer, you override the `layerClass` method in the custom table view cell class to return your gradient layer.

Handle that final step by adding the following code to **BTThemeTableViewCell.m**:

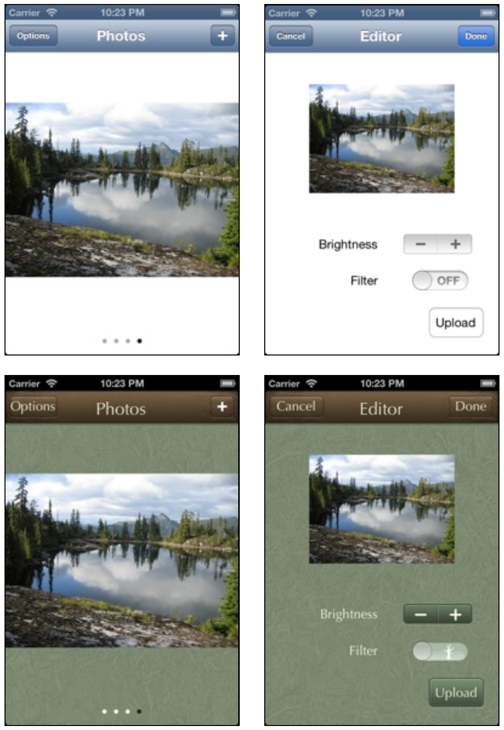
```
#import "BTTheme.h"

+(Class)layerClass{
    return [[BTThemeManager sharedTheme] gradientLayer];
}
```

Build and run, and you can see the new styling for the table view cells when using the Forest theme.



Speaking of which, what is this Pretty in Pink theme you see in the table view? It's about time to find out.

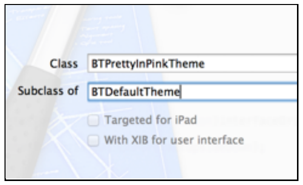
Before you move on though, spend a moment enjoying your fully-customized app:



Adding your own theme

Now that you've customized the entire application and have the theme manager infrastructure set up and implemented, I'd like to show you how easy it is to add a completely new theme.

Control-click on the **Theme** group and select **New File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, and click **Next**. Name the class **BTPrettyInPinkTheme**, make it a subclass of **BTDefaultTheme**, and click **Next** and finally **Create**.



Class BTPrettyInPinkTheme
Subclass of BTDefaultTheme
Targeted for iPad
With XIB for user interface

Paste the following into BTPrettyInPinkTheme.m (kids, do not try to type all of this at home!):

```
#import "BTPrettyInPinkTheme.h"

@implementation BTPrettyInPinkTheme
- (UIColor*)backgroundColor {
    return [UIColor colorWithPatternImage:
            [UIImage imageNamed:@"bg_prettyinpink.png"]];
}

- (UIColor*)upperGradient{
    return [UIColor colorWithRed:0.961 green:0.878
                        blue:0.961 alpha:1.000];
}
- (UIColor*)lowerGradient {
    return [UIColor colorWithRed:0.906 green:0.827
                        blue:0.906 alpha:1.000];
}
- (UIColor*)seperatorColor {
    return [UIColor colorWithRed:0.871 green:0.741
                        blue:0.878 alpha:1.000];
}

- (UIColor*)progressBarTintColor {
    return [UIColor colorWithRed:0.600 green:0.416
                        blue:0.612 alpha:1.000];
}
- (UIColor*)progressBarTrackTintColor {
    return [UIColor colorWithRed:0.749 green:0.561
                        blue:0.757 alpha:1.000];
}

- (UIColor*)switchOnTintColor {
    return [UIColor colorWithRed:0.749 green:0.561
                        blue:0.757 alpha:1.000];
}
```

```
- (UIColor*)switchThumbTintColor {
    return [UIColor colorWithRed:0.918 green:0.839
                            blue:0.922 alpha:1.000];
}

- (UIColor*)pageTintColor {
    return [UIColor colorWithRed:0.290 green:0.051
                            blue:0.302 alpha:1.000];
}
- (UIColor*)pageCurrentTintColor {
    return [UIColor colorWithRed:0.749 green:0.561
                            blue:0.757 alpha:1.000];
}

- (UIImage*)imageForBarButtonNormal {
    return [[UIImage imageNamed:@"barbutton_pretty_uns.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 9.0, 0.0, 9.0)];
}
- (UIImage*)imageForBarButtonHighlighted {
    return [[UIImage imageNamed:@"barbutton_pretty_sel.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 9.0, 0.0, 9.0)];
}
- (UIImage*)imageForBarButtonDoneNormal {
    return [[UIImage imageNamed:
                     @"barbutton_pretty_done_uns.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 9.0, 0.0, 9.0)];
}
- (UIImage*)imageForBarButtonDoneHighlighted {
    return [[UIImage imageNamed:
                     @"barbutton_pretty_done_sel.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 9.0, 0.0, 9.0)];
}

- (UIImage*)imageForBarButtonNormalLandscape {
    return [[UIImage imageNamed:
                     @"barbutton_pretty_landscape_uns.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 7.0, 0.0, 8.0)];
}
- (UIImage*)imageForBarButtonHighlightedLandscape {
    return [[UIImage imageNamed:
```

```
                     @"barbutton_pretty_landscape_sel.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 7.0, 0.0, 8.0)];
}
- (UIImage*)imageForBarButtonDoneNormalLandscape {
    return [[UIImage imageNamed:
                     @"barbutton_pretty_done_landscape_uns.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 7.0, 0.0, 8.0)];
}
- (UIImage*)imageForBarButtonDoneHighlightedLandscape {
    return [[UIImage imageNamed:
                     @"barbutton_pretty_done_landscape_sel.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 7.0, 0.0, 8.0)];
}

- (UIImage*)imageForButtonNormal {
    return [[UIImage imageNamed:@"button_pretty_uns.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 13.0, 0.0, 13.0)];
}
- (UIImage*)imageForButtonHighlighted {
    return [[UIImage imageNamed:@"button_pretty_sel.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 13.0, 0.0, 13.0)];
}

- (UIImage*)imageForNavigationBar{
    return [[UIImage imageNamed:@"nav_pretty_portrait.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 12.0, 0.0, 12.0)];
}
- (UIImage*)imageForNavigationBarLandscape{
    return [[UIImage imageNamed:@"nav_pretty_landscape.png"]
            resizableImageWithCapInsets:
            UIEdgeInsetsMake(0.0, 12.0, 0.0, 11.0)];
}
- (UIImage*)imageForNavigationBarShadow{
    return [UIImage imageNamed:@"topShadow_pretty.png"];
}

- (UIImage*)imageForSwitchOn{
    return [UIImage imageNamed:@"floweron.png"];
}
```

```
- (UIImage*)imageForSwitchOff{
    return [UIImage imageNamed:@"floweroff.png"];
}

- (UIImage*)imageForStepperUnselected{
    return [UIImage imageNamed:@"stepper_pretty_bg_uns.png"];
}
- (UIImage*)imageForStepperSelected{
    return [UIImage imageNamed:@"stepper_pretty_bg_sel.png"];
}
- (UIImage*)imageForStepperDecrement{
    return [UIImage imageNamed:@"stepper_pretty_decrement.png"];
}
- (UIImage*)imageForStepperIncrement{
    return [UIImage imageNamed:@"stepper_pretty_increment.png"];
}
- (UIImage*)imageForStepperDividerUnselected{
    return [UIImage imageNamed:
                    @"stepper_pretty_divider_uns.png"];
}
- (UIImage*)imageForStepperDividerSelected{
    return [UIImage imageNamed:
                    @"stepper_pretty_divider_sel.png"];
}

- (NSDictionary*)navBarTextDictionary {
return @{
UITextAttributeFont:
    [UIFont fontWithName:@"Arial Rounded MT Bold" size:18.0],
UITextAttributeTextColor:
    [UIColor colorWithRed:0.290 green:0.051
                    blue:0.302 alpha:1.000],
UITextAttributeTextShadowColor:
    [UIColor colorWithRed:0.965 green:0.945
                    blue:0.965 alpha:1.000],
UITextAttributeTextShadowOffset:
    [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
    };
}
- (NSDictionary*)barButtonTextDictionary  {
return @{
UITextAttributeFont:
    [UIFont fontWithName:@"Arial Rounded MT Bold" size:15.0],
UITextAttributeTextColor:
    [UIColor colorWithRed:0.506 green:0.314
```

```
                    blue:0.510 alpha:1.000],
UITextAttributeTextShadowColor:
    [UIColor colorWithRed:0.965 green:0.945
                    blue:0.965 alpha:1.000],
UITextAttributeTextShadowOffset:
    [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
    };
}
- (NSDictionary*)buttonTextDictionary  {
return @{
UITextAttributeFont:
    [UIFont fontWithName:@"Arial Rounded MT Bold" size:18.0],
UITextAttributeTextColor:
    [UIColor colorWithRed:0.290 green:0.051
                    blue:0.302 alpha:1.000],
UITextAttributeTextShadowColor:
    [UIColor colorWithRed:0.965 green:0.945
                    blue:0.965 alpha:1.000],
UITextAttributeTextShadowOffset:
    [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
    };
}
- (NSDictionary*)labelTextDictionary  {
return @{
UITextAttributeFont:
    [UIFont fontWithName:@"Arial Rounded MT Bold" size:18.0],
UITextAttributeTextColor:
    [UIColor colorWithRed:0.290 green:0.051
                    blue:0.302 alpha:1.000],
UITextAttributeTextShadowColor:
    [UIColor colorWithRed:0.965 green:0.945
                    blue:0.965 alpha:1.000],
UITextAttributeTextShadowOffset:
    [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
    };
}

- (NSDictionary*)tableViewCellTextDictionary  {
return @{
UITextAttributeFont:
    [UIFont fontWithName:@"Arial Rounded MT Bold" size:18.0],
UITextAttributeTextColor:
    [UIColor colorWithRed:0.290 green:0.051
                    blue:0.302 alpha:1.000],
UITextAttributeTextShadowColor:
    [UIColor colorWithRed:0.965 green:0.945
                     blue:0.965 alpha:1.000],
UITextAttributeTextShadowOffset:
    [NSValue valueWithUIOffset:UIOffsetMake(0, 1)]
    };
}
@end
```

Yes, that's the full theme implementation. ☺ If you look at the code, you'll realize that this is identical to what you did for the Forest theme – just the images, colors, and fonts are different.

So you can see that once you have one theme done, it's a fairly simple matter to implement additional themes by changing style elements.

Of course, you still need to add the hooks in the theme controller to allow selection of the theme.

In **BTThemeViewController.m**, add the following import at the top:

```
#import "BTPrettyInPinkTheme.h"
```

And add the following code to `tableView:didSelectRowAtIndexPath:` (right before the `else` statement):

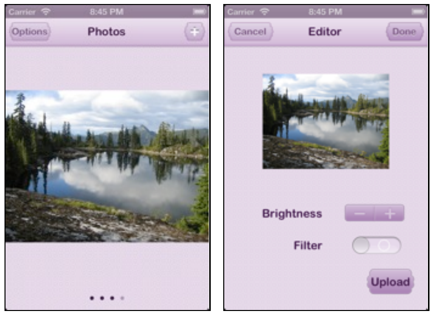
```
    else if ([self.themes[idx]
                           isEqualToString:@"Pretty in Pink"]) {
      [BTThemeManager setSharedTheme:[BTPrettyInPinkTheme new]];
    }
```

The final method should look like this:

```
-(void)tableView:(UITableView *)tableView
              didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
```

```
    NSInteger idx = indexPath.row;

    if ([self.themes[idx] isEqualToString:@"Forest"]) {
      [BTThemeManager setSharedTheme:[BTForestTheme new]];
    }
    else if ([self.themes[idx]
                              isEqualToString:@"Pretty in Pink"]) {
      [BTThemeManager setSharedTheme:[BTPrettyInPinkTheme new]];
    }
    else
      [BTThemeManager setSharedTheme:[BTDefaultTheme new]];

    [self.navigationController popViewControllerAnimated:YES];
  }
```

Build and run, go to Options, and select the "Pretty in Pink" theme.



Oooh, pretty, huh? ☺

Where to go from here?

At this point you have learned about all the new user interface customization options in iOS 6, as well as a bunch of other cool stuff, like creating a dynamic theme manager and using gradient alyers!

What's most important is that you understand the theme manager and how to create new customizations. As you have seen, after your manager is set up, adding a completely new theme is comparatively simple.

I would encourage you to take one of the themes and:

• Play with it and modify it a bit.

• Use it as a template to create a brand new theme.

• Add a new type of control not covered here and learn how to customize it in your theme.

For more information, I encourage you to watch session #216 from the WWDC 2012 developer videos, "Advanced Appearance Customization on iOS."

Have fun theming!