Extracted from:

Beginning Mac Programming

Develop with Objective-C and Cocoa

The Pragmatic Programmers

Beginning Mac Programming

Develop with Objective-C and Cocoa

Tim Isted

Edited by Colleen Toporek

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-51-4

ISBN-13: 978-1-934356-51-7

Printed on acid-free paper.

P1.0 printing, March 2010

Version: 2010-5-16

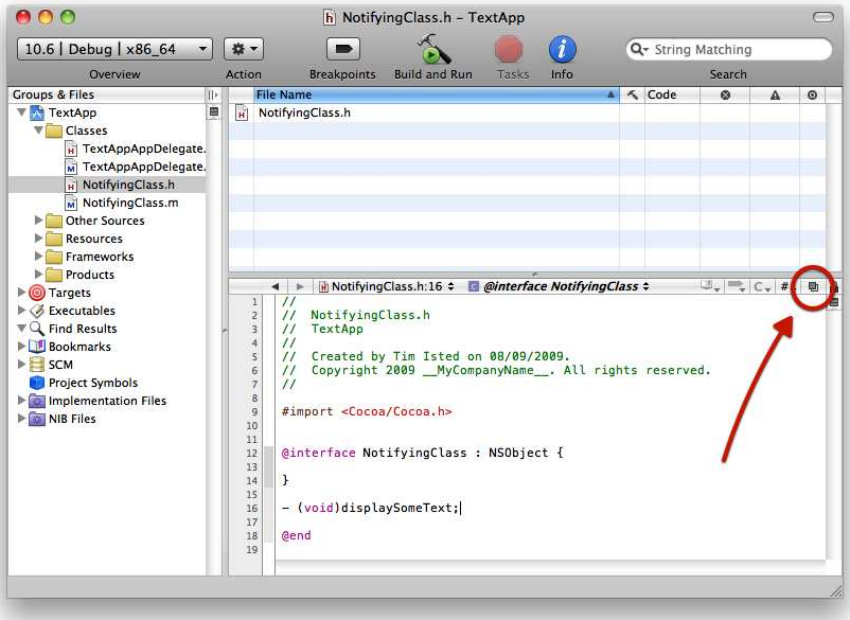

Figure 4.1: The Go to Counterpart quick-switch icon in Xcode

For now, we'll just output a message to Xcode's debugger console to let us know when this method is called, so add the following line of code:

```
- (void)displaySomeText
{
    NSLog(@"displaySomeText just got called!");
}
```

If you Build & Run at this point, you'll find that TextApp launches fine, but no friendly messages appear in the debugger console. We removed the code responding to the init message earlier, and at the moment our new displaySomeText method never gets called. All we've done so far is define the code that would be called *if* the displaySomeText message was received, but it never is.

4.2 The Target-Action Mechanism

Our aim is to send the displaySomeText message to our NotifyingClass object when the user clicks a button in our TextApp application. The

Cocoa framework provides us with a nifty little technique to help with this, called *target-action*.

Certain objects offered by the Cocoa framework allow you to provide them with a *target* object and specify an *action*—a message to be sent to that object. One of the objects supporting such a mechanism is an NSButton. This class is used to create a number of different types of Mac OS X buttons; its default style is the one we'll be using shortly when we add a button to TextApp's user interface. Once we've created an NSButton instance in Interface Builder, we can tell the button about our NotifyingClass instance and specify that the relevant *action* is to call its displaySomeText method when the user clicks the button.

At this point we need to make a couple of small changes to our existing method definition for displaySomeText so we can use it with Interface Builder and target-action. For Cocoa desktop applications on the Mac, a method to be called in a target-action situation needs to conform to a specific signature format. The method needs to be able to accept a single *argument* on it.

We'll discuss arguments in Chapter 6, *Passing Information Around*, on page 99, but if you take a quick look back now to earlier in this chapter at the format for a message signature, you'll see that there are "optional bits" that can be put on the end of the signature style we've seen so far, as in one example from NSObject.h:

```
+ (id)allocWithZone:(NSZone *)zone;
```

If a message name has a colon on the end of it (that's : rather than the end-of-line *semi*colon ;), it means that it accepts one or more arguments. An argument is used to pass some information along with a message. Back in our housing example, when the property developer object sends a message to the builder object to buildHouse:, it needs to specify which house to build. This would be offered as an argument, like this:

```
- (void)buildHouse:(House *)houseToBeBuilt;
```

After the colon comes something in parentheses with an asterisk—(House *)—that specifies what kind of information is being provided, followed by a name to be assigned to the provided information. Don't worry if this isn't immediately clear in your head; we'll be covering various things in the next chapter that will help you to understand it. We'll continue using it in this chapter, though, so at this point, just follow along with the syntax.

```
+ (id)                          allocWithZone:(NSZone *)            zone;
- (void)                        buildHouse: (House *)               houseToBeBuilt;
± («type expected in return»)methodName:(«type of info provided»)«name for info»;
```

The buildHouse: method (notice we're putting the colon on the end of it to indicate it accepts an argument) expects to receive a House object when it's called, identifying the particular house to be built. Similarly, the allocWithZone: method expects an NSZone object to identify the zone to be used.

Returning to target-action, I stated a little earlier that a method needs to accept a single argument to work for target-action. This argument happens to be used to identify the object that sent the message in the first place. When we connect a button in TextApp to target our NotifyingClass instance, the *button* would send the message and specify that *it* was the sender. The signature therefore needs to look something like this:

```
- (void)displaySomeText:(id)sender;
```

The **id** keyword here is used to specify that *some kind of object* will be provided as the sender. Again, we'll be talking more about **id** later.

There's just one last thing we need to change slightly on this method declaration, since we're going to be using Interface Builder to connect the target and action on a new NSButton object.

Communication Between Xcode and Interface Builder

Remember how we created an object instance in Interface Builder (that blue cube thing) in the previous chapter and set its class to NotifyingClass in the inspector? When we did that, Interface Builder managed to guess the rest of the class name after we'd typed only a few characters. This is because Interface Builder and Xcode "talk to each other" behind the scenes. Some things happen automatically (like IB having knowledge of classes for which you've provided class descriptions in an Xcode project), whereas other things happen when you use specific keywords in your code files.

This is one case where we need to use such a keyword. To specify that our method is an action that can be connected to a control object (like an NSButton) in Interface Builder, we need to change the **void** keyword into **IBAction**:

```
- (IBAction)displaySomeText:(id)sender;
```

As far as the *code compiler* is concerned, **IBAction** is synonymous with **void**. This keyword simply alerts Interface Builder that there's an *action* available in a class description.

It's time to modify our code for this purpose. First, we need to change the interface for our NotifyingClass object, so open NotifyingClass.h in Xcode. Change it so that the method follows our new action signature:

```
@interface NotifyingClass : NSObject {
}

- (IBAction)displaySomeText:(id)sender;

@end
```

Next, we need to change our implementation method signature in NotifyingClass.m, so switch to this file, and make it look like this:

```
@implementation NotifyingClass

- (IBAction)displaySomeText:(id)sender
{
    NSLog(@"displaySomeText just got called!");
}

@end
```

Notice that we don't need to do anything with the sender attribute—we can quite safely ignore it altogether. And, since **IBAction** is just a keyword meaning **void** as far as the compiler is concerned, we don't have to worry about giving any information back at the end of the method.

Make sure you save all your modifications before proceeding. If you hold down the ⌥ key on your keyboard and go to Xcode's File menu, you'll see that a few of the items have changed, and there is now a Save All... command (with keyboard shortcut ⌥-⌘-S.) If there are any unsaved changes in any of the project's files, use this command to save them all at once.

Connecting the Action in Interface Builder

Now that we've defined our NotifyingClass class instances to respond to an action message for displaySomeText, we'll use Interface Builder to connect things visually.

Let's start by adding a button to our existing TextApp window. Open MainMenu.xib, and make sure TextApp's Window object with the text view is visible.

Figure 4.2: The new push button in our Text App main window

You will need to make that text view a little smaller by dragging up the bottom handle so there's room for a button underneath it. Next, find a standard Push Button object in the Library palette, and drag one out onto the window. You should end up with something resembling Figure 4.2.

Next, remove one of the existing NotifyingClass instances from the MainMenu.xib file by selecting it and pressing the ⌫ key.

We're now ready to connect our button to the remaining NotifyingClass instance, and Interface Builder provides a great interface for doing this. Bring the MainMenu.xib document window to the front in Interface Builder, but keep the window with the button in it visible on screen. Right-click (or ^-click) the Notifying Class blue cube. You'll find that a black mini-window pops up (known as a *heads-up display*, or HUD), as shown in Figure 4.3, on the following page.

Figure 4.3: The connections HUD for our NotifyingClass instance

If everything has gone to plan, you should find that this HUD window shows one option under the heading "Received Actions"—our displaySomeText: action message. To the right of this line is a little circle; click and drag from the circle, and you'll find a line extends out and follows your movements on screen. Drag down to our new Button object, and you'll find IB highlights and identifies the button. Release your mouse, and you should find that the HUD display now shows a connection to our Push Button, as in Figure 4.4, on the next page.

If you now bring the application's window to the front, click the push button to select it, and then right-click (or ^-click) on it, you'll find that another HUD window will appear, showing a connection to our NotifyingClass object's action under the "Sent Actions" heading, as in Figure 4.5, on page 68.

It really is as simple as that! Interface Builder has now set up our button with a reference to the notifying object and has set up the requested action for us, all with just a simple click-and-drag motion.

Save your file in Interface Builder, and switch back to Xcode to Build & Run. TextApp will launch, and when you click the button in the window, you'll see a message appear each time in the Xcode console window. Woo-hoo!

4.3 Sending Messages from Our Code

The target-action mechanism we explored in the previous section is awesome for connecting up buttons and *receiving* messages in certain situations, but we need to *send* messages to objects from our code too.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Beginning Mac Programming's Home Page
http://pragprog.com/titles/tibmac
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/tibmac.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)