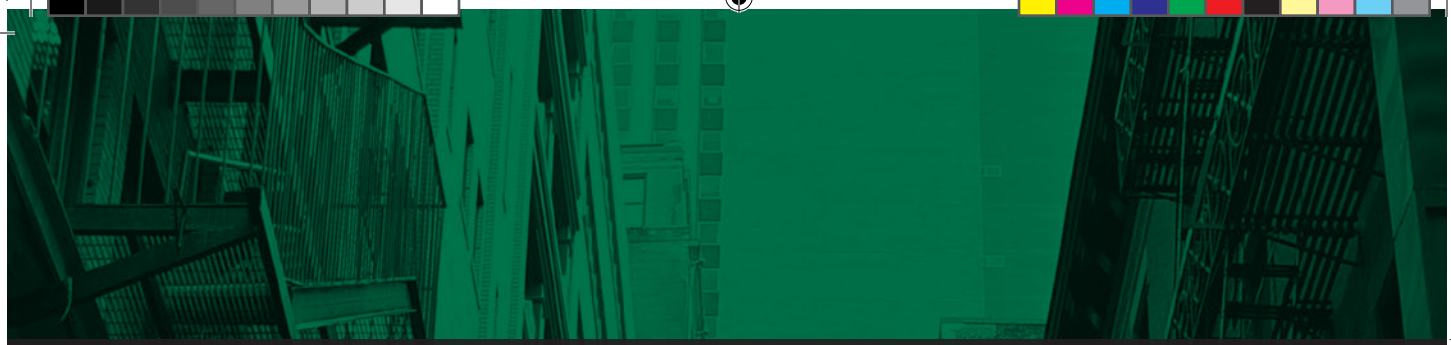
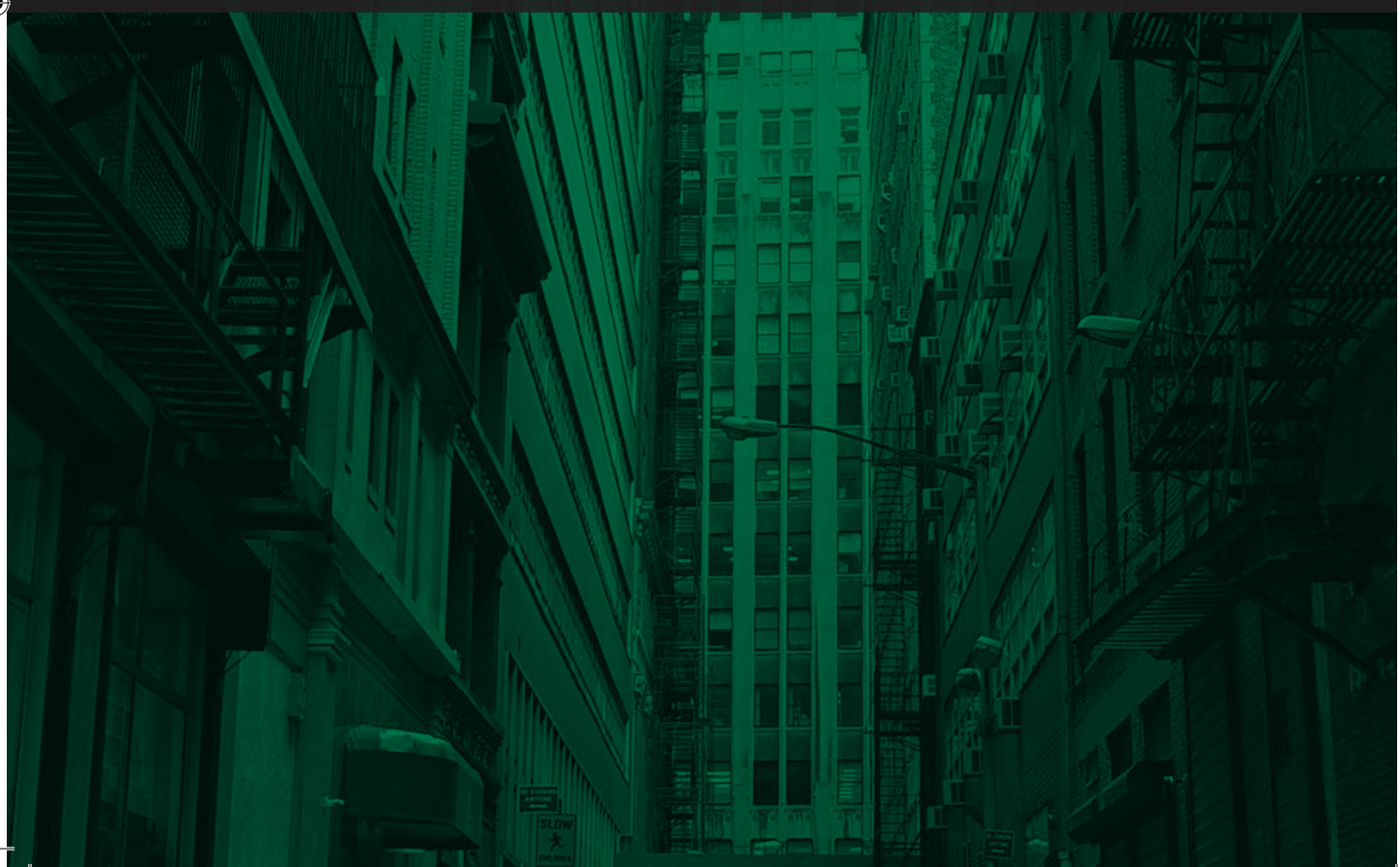


BONUS

Overview of iOS Notification Techniques



We've seen a number of notification techniques so far. It's probably worth doing a quick review over each approach. They all have the same basic goal—trying to balance the twin pressures of creating modular, loosely bound code that is easy to maintain, with the need to pass information across these boundaries. However, the different approaches have differing strengths and weaknesses.

CALLING METHODS ON PROPERTIES

Not to be overlooked, this is the simplest and most direct approach. It is also the de facto communication channel used inside a module. You create a property that stores a reference to an object, and then you just call that object's methods directly.

Unfortunately, direct method calls tends to tightly bind the objects together. We can use a few software engineering techniques to help minimize these bonds—for example, using protocols to define the interface between modules and then coding to the protocol, instead of coding to the particular objects. However, this approach will never be as loosely bound as some of the alternatives.

Additionally, this creates a one-direction communication channel. We typically use properties to send messages down the object graph. Making links back up the graph gets tricky, since these references can create retain cycles if not handled properly. This means we will often use other techniques (especially delegate-like interfaces) to communicate back up the graph.

OUTLETS

An outlet is simply a property that is defined with the `IBOutlet` keyword. This lets us draw connections to the outlet in Interface Builder. Outlets typically use `weak` storage, since they usually refer to items in the view hierarchy. These objects will be kept in memory by their superview. However, if we are referring to objects either that are not in the view hierarchy or that may be removed from the view hierarchy (such as our input accessories in Chapter 3, "Developing Views and View Controllers"), we need to use a `strong` reference instead.

KEY-VALUE OBSERVATION

This lets us receive notifications when an object's property changes. This is particularly useful for detecting changes when no official notification method exists. Here, we don't need to wait for someone else to create the communication channel for us. It is built in to the underlying object model.

KVO promotes loose binding between the observed and the observer. The observed object doesn't need to know who or what is observing it. There could be zero, one, or a million observers. The observed object does not care. Additionally, the observed object doesn't need to do anything to generate the notifications—the system dispatches them automatically.

Unfortunately, KVO notifications have a few shortcomings. First, all notification methods are channeled through the `observeValueForKeyPath:ofObject:change:context:` method, which makes monitoring more than a few notifications somewhat cumbersome.

Second, we must make sure our objects are KVO and KVC compliant. This is easy for most properties, but virtual and container properties require a bit of special care and attention.

Third, KVO notifications can cause unexpected side effects. Once you start using KVO notifications, any call that assigns a new value to a property could also trigger arbitrary method calls elsewhere in our code. This is one of the reasons we tend to avoid using properties inside the `dealloc` method.

Finally, we need to make sure we remove our observers before they are deallocated. Otherwise, we could leave dangling pointers behind, which could crash the application.

TARGET-ACTION

The target-action pattern is specifically used by `UIControl` and its subclasses. It lets an event trigger an action on a given target. We typically use target-actions to let a control communicate back to its view controller, usually in response to user interactions. Unlike other notification methods, we don't usually worry about unregistering target-action events

While very powerful, it is also limited to a few, special-use cases. Specifically, `UIControl` defines a limited number of events. We cannot add our own, custom events. We are also limited by the action method signature, which can take zero, one, or two predefined arguments.

One of the biggest strengths of target-action is that action methods that are defined with the `IBAction` keyword can be connected in Interface Builder. The target-action pattern also supports triggering the same action from multiple events or controls. It even supports triggering multiple actions from the same control and event.

We typically use predefined `UIControl` subclasses; however, we can also create our own `UIControl` subclasses. We just need to call `sendActionsForControlEvents:` whenever a given event occurs.

DELEGATE-LIKE INTERFACES

We can create delegates, data sources, and related objects to provide a very rich interface between two objects. Delegates use weak references to avoid retain cycles, which makes them useful when communicating back up the object graph. This also means we don't need to worry about removing a delegate—since it will automatically be zeroed out when the delegate is deallocated.

We often use delegates to communicate between user interface elements and the view controller when simple target-action methods don't suffice. In the past, we also used delegates to communicate between different view controllers, especially between presented and presenting view controllers. However, that has been largely replaced by segue unwinding in modern storyboard applications.

Delegates can also be used instead of subclassing, letting us monitor and modify the behavior of existing classes. Even more generally, delegates are a great way to create an arbitrarily complex communication channel between two otherwise loosely bound objects.

Delegates are typically one-to-one communication channels, though it is possible to create one-to-many channels, like our subscribers.

Their only real weakness is their complexity. We often need to research a delegate's protocol before we can use it properly. And, if we are creating our own delegate-like interface, there's a fair bit of work involved in defining the protocol, setting up and managing the storage, and calling the delegate methods.

Another disadvantage is that sometimes our application's logic can get spread out across a number of delegate methods. This can make it hard to follow the application's flow, if these methods are not well organized.

In many modern APIs, block-based callback handlers have begun replacing delegates.

CALLBACK HANDLERS

A growing number of methods take a callback handler (or other completion block). This is a block that is triggered under specific situations. These are most often used in asynchronous APIs. The typical pattern is that we call the method on the main thread and provide a callback block. The method does its work on a background thread. When it is done, it calls our callback block on the main thread.

Callback handlers are also being used as delegate replacements. Instead of passing a delegate object whose methods get called, you just pass a block, which gets called directly. While this doesn't scale as well as the delegate-like interface, it helps keep our code organized in one location. This is particularly useful when working on smaller, more focused communication channels.

NOTIFICATION CENTER

This is the true workhorse for loosely bound interfaces. The sending object and the receiving object never interact directly. Instead, the notifications are routed through the notification center. The sender simply posts notifications to the center. It doesn't know who, if anyone, is listening.

On the other side, the listener simply registers with the notification center to receive notifications. It doesn't even need to know which object is sending the notification messages. It just needs the notification's name.

Notifications don't typically carry a lot of data. While we can put some information in the `userInfo` dictionary, we usually send simple notification messages and expect the user to

request any additional information in response. This often leads to a proliferation of notification types, with a different notification for each type of change that might occur.

Also, while notifications can be used to send messages to any arbitrary number of listeners, this quickly leads to a considerable amount of duplicate or boilerplate code. Therefore, notifications might not be the best solution for widely broadcasted notifications.

Finally, just like KVO, we need to make sure we unregister our observers before they are deallocated.

PASSING MESSAGES DOWN HIERARCHIES

We frequently see this approach used in our view and view controller hierarchies. Hit testing, drawing, layout updates, and similar messages are all dispatched down the view hierarchy. Rotation messages are sent down the view controller hierarchy. The responder chain and the `NSCoding` protocol are just a variation on this theme.

For this to work, the hierarchy must contain objects that are all subclasses of the same class or objects that all adopt the same protocol. For example, everything in the view hierarchy is a subclass (directly or indirectly) of `UIView`. Similarly, everything in the responder chain is a subclass of `UIResponder`.

The common ancestor defines the notification method that gets chained along. We override this method in our custom subclass to respond to the notification—usually calling `super` at some point, passing the method on down the chain.

This is a great way to broadcast notifications to a large number of similar objects. However, implementing our own broadcasts often means adding custom methods to existing classes like `UIView`, `UIViewController`, or even `NSObject`. This requires a bit of care and thought—since it could have a broad impact across large parts of our application.

WRAPPING UP

As you can see, we have a number of tools to help us build modular, loosely bound code. In fact, many of these notification techniques have use cases that largely overlap. Unfortunately, there are no clear-cut rules for selecting one particular approach over the others. Often it just comes down to personal preference. Other times, the choice will be dictated by the needs of the project. Using a consistent approach to notification can make a project easier to understand and maintain. Often, this is more important than picking the exact best solution for a particular situation.

Still, I feel that a good command of the entire range of notification techniques is important. You should be able to both understand and implement each of these approaches, even if you only end up using a subset in your own projects.