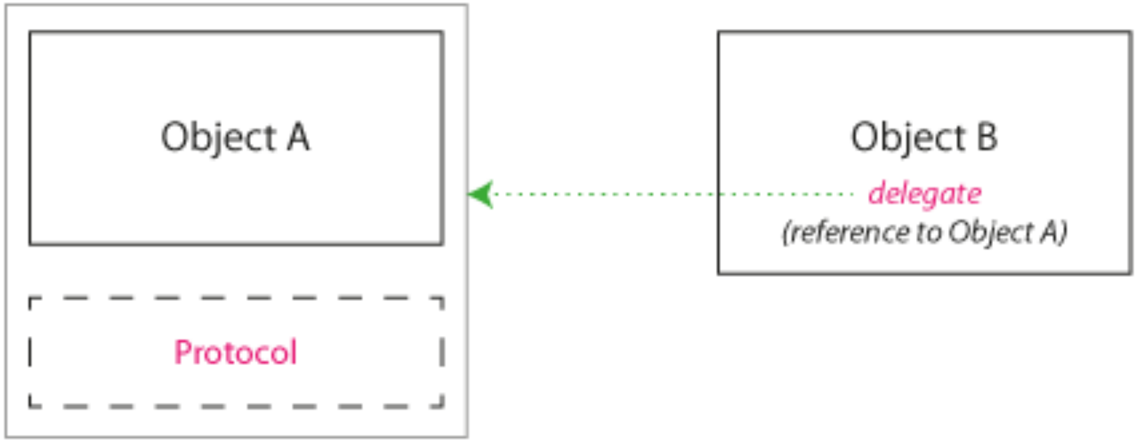
ITP 342 Mobile App Dev



Delegates

Delegation

- **Delegation** (or using delegates) is one of Cocoa’s most significant design patterns.
- A **delegate** is an object that agrees to undertake certain decisions or tasks for another object, or would like to be notified when certain events occur.
- An object can have a single **delegate** object that it will call when certain events occur.
  - From the delegate's point of view, it's like a callback.
  - From the delegating object's point of view, it's like handing off responsibility.

Delegates

- A **delegate** is an object that **acts on behalf of**, or in coordination with, **another object** when that object encounters an event in a program.
- The delegating object is often a responder object that is responding to a user event.
  - An example is an object inheriting from UIResponder in UIKit.
- The delegate is an object that is delegated control of the user interface for that event, or is at least asked to interpret the event in an application-specific manner.

Delegates

- The programming mechanism of delegation gives objects a chance to coordinate their appearance and state with changes occurring elsewhere in a program, changes usually brought about by user actions.
- More importantly, delegation makes it possible for one object to alter the behavior of another object without the need to inherit from it.
- The delegate is almost always one of your custom objects, and by definition it incorporates application-specific logic that the generic and delegating object cannot possibly know itself.

Delegation

- The delegating object keeps a reference to the other object – the delegate – and at the appropriate time sends a message to it.
  - The message informs the delegate of an event that the delegating object is about to handle or has just handled.
  - The delegate may respond to the message by updating the appearance or state of itself or other objects in the application, and in some cases it can return a value that affects how an impending event is handled.
- The main value of delegation is that it allows you to easily customize the behavior of several objects in one central object.

Becoming a Delegate

- Three-step process:
  1. In your custom class, adopt the delegate's protocol.
  2. Implement the appropriate protocol methods.
  3. Connect the `delegate` outlet of the object to your delegate object.

Protocol

- A **protocol** is a contract, or promise, that your class will implement specific methods.
- This lets other objects know that your object has agreed to accept certain responsibilities.
- A protocol can declare two kinds of methods: required and optional.
  - All **required** methods must be included in your class's implementation.
    - If you leave any out, you've broken the contract, and your project won't compile.
  - If you implement an **optional** method, your object will receive that message.
    - If you don't, it won't.
  - Most delegate methods are optional.

Delegates

- Cocoa Touch makes extensive use of delegates
  - Classes that take responsibility for doing certain things on behalf of another object
- Every iOS application has one and only one instance of `UIApplication`, which is responsible for the application's run loop and handles application-level functionality such as routing input to the appropriate controller class
  - `UIApplication` will call specific delegate methods, if there is a delegate and if it implements that method
  - For example, if you have code that needs to fire just before your program quits, you would implement the method `applicationWillTerminate:` in your application delegate and put your termination code there

Adopt the Delegate's Protocol

- Add the <DelegateName> to the interface file OR class extension of implementation file

```
// ViewController.h
@interface ViewController: UIViewController <DelegateName> {
...
}
...
@end
```

```
// ViewController.m
@interface ViewController () <DelegateName>
@end

@implementation ViewController
...
@end
```

Adopt Multiple Protocols

- You may adopt multiple protocols

```
// ViewController.h
@interface ViewController: UIViewController <Delegate1, Delegate2> {
...
}
...
@end
```

```
// ViewController.m
@interface ViewController () <Delegate1, Delegate2>
@end

@implementation ViewController
...
@end
```

Delegation and the Cocoa Frameworks

- The delegating object is typically a framework object, and the delegate is typically a custom controller object.
- In a managed memory environment, the delegating object maintains a weak reference to its delegate; in a garbage-collected environment, the receiver maintains a strong reference to its delegate.
- Examples of delegation abound in the Foundation, UIKit, AppKit, and other Cocoa and Cocoa Touch frameworks.

Delegation and Notification

- The delegate of most Cocoa framework classes is automatically registered as an observer of notifications posted by the delegating object.
- The delegate need only implement a notification method declared by the framework class to receive a particular notification message.

Data Source

- A data source is almost identical to a delegate.
- The difference is in the relationship with the delegating object.
- Instead of being delegated control of the user interface, a data source is delegated control of data.
- The delegating object, typically a view object such as a table view, holds a reference to its data source and occasionally asks it for the data it should display.
- A data source, like a delegate, must adopt a protocol and implement at minimum the required methods of that protocol.
- Data sources are responsible for managing the memory of the model objects they give to the delegating view.

Summary

- Delegation can be used in a broad range of situations, from simple callbacks to the support of complex user interface elements like table views.
- In doing so delegation helps achieve the goal of allowing objects to work with little interaction between them, allowing some objects to be relatively fixed and others highly customized.
- The result supports the goal of maximum software reuse and the Model-View-Controller (MVC) design pattern.