View Controller Programming Guide for iOS

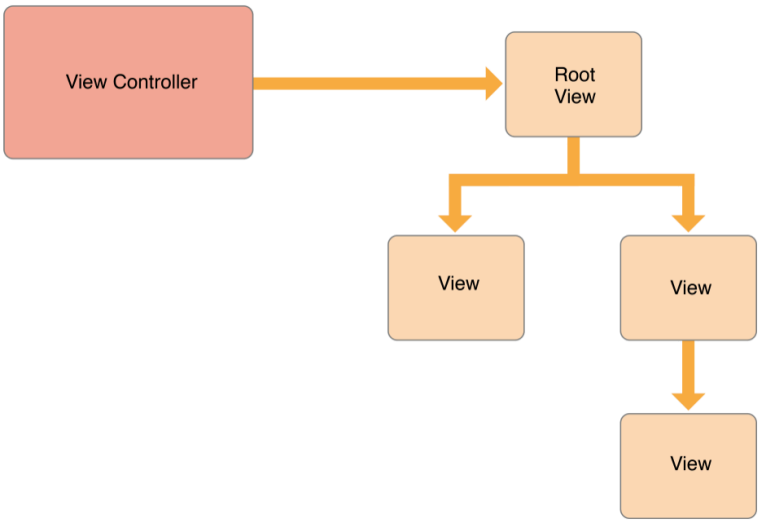
[https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/index.html#//apple_ref/doc/uid/TP40007457]

---

1. Overview

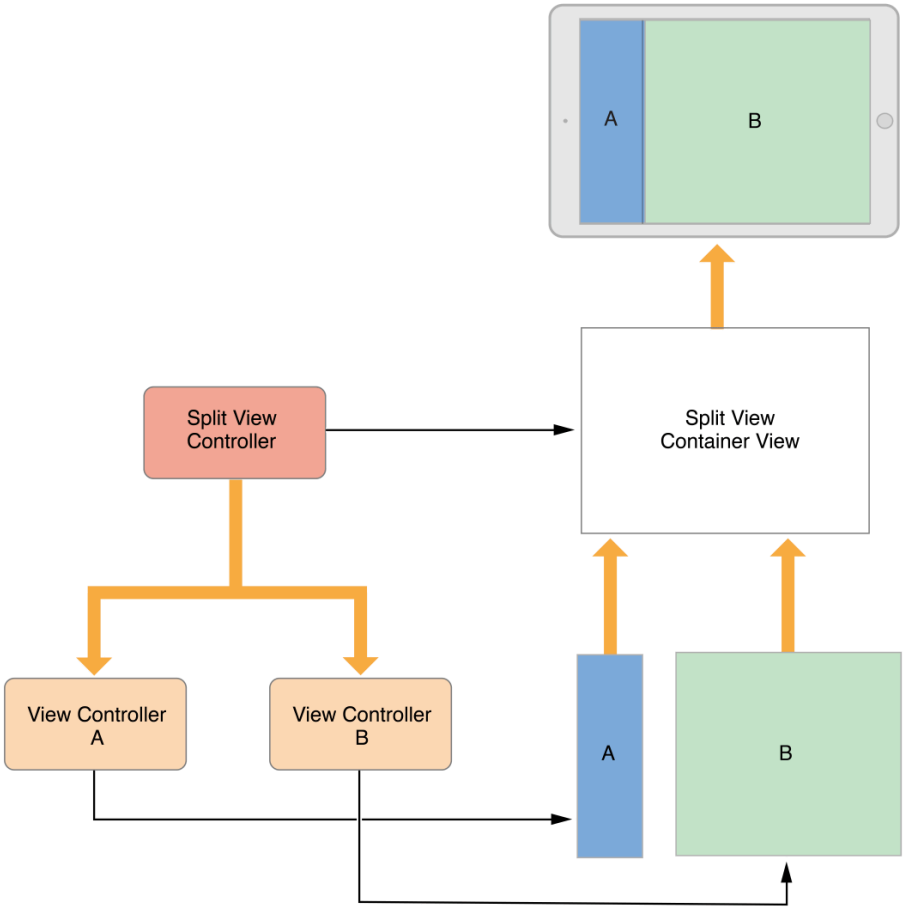
- The Role of View Controllers
  UIViewController class:
  1. manage portion of app's user interface, interaction between interface and underlying data, and handling events.
  2. facilitate transitions between different parts of user interface
     - two types:
       - 1. Content view controllers.
       - 2. Container view controllers.
     - View Management:
       - content view controller manage all of its views by itself.
       - container view controller manage its own views + root views from its child view controllers.

Figure 1-1 Relationship between a view controller and its views



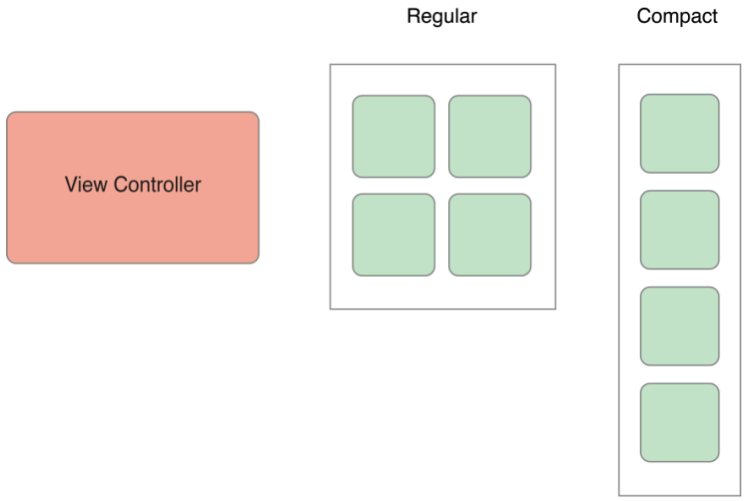
-

Figure 1-2 View controllers can manage content from other view controllers



- Data Marshaling
  - separation view controller and data object, like dataModel, view, controller handle logic.
- User Interaction
  - handle touch events — usually view(not view controller) with associated delegate/ target object(view controller).
  - View Controller typically handle events in following ways:
    - Action methods — @IBAction, #selector, target-action, gesture recognizer
    - Observe notifications by system/other objects.
    - Act data source/delegate for another object. like: table view, collection view.
- Resource Management
  - view controller call `didReceiveMemoryWarning` method remove reference to objects like: cached data
- Adaptivity
  - horizontal trait, vertical trait size: regular, compact.

Figure 1-4 Adapting views to size class changes



- The Adaptive Model:
  - 1. (Auto-Layout: define constraints)
  - 2. traits (high-level decision about interface)
    - Role of Traits:

Table 12-1 Traits

Trait	Examples	Description
`horizontalSizeClass`	`UIUserInterfaceSizeClassCompact`	This trait conveys the general width of your interface. Use it to make coarse-level layout decisions, such as whether views are stacked vertically, displayed side by side, hidden altogether, or displayed by another means.
`verticalSizeClass`	`UIUserInterfaceSizeClassRegular`	This trait conveys the general height of your interface. If your design requires all of your content to fit on the screen without scrolling, use this trait to make layout decisions.
`displayScale`	`2.0`	This trait conveys whether the content is displayed on a Retina display or a standard-resolution display. Use it (as needed) to make pixel-level layout decisions or to choose which version of an image to display.
`userInterfaceIdiom`	`UIUserInterfaceIdiomPhone`	This trait is provided for backward compatibility and conveys the type of device on which your app is running. Avoid using this trait as much as possible. For layout decisions, use the horizontal and vertical size classes instead.

  - 1. Use size class to make coarse changes to interface.
  - 2. Never assume that a size class corresponds to specific width/height of a view.
  - 3. Use IB to specify different layout constraints for each size

class.
    - 4. avoid using idiom information to make decision.
- - When trait and size changes?

Figure 12-2 Updating a view controller's traits and view size

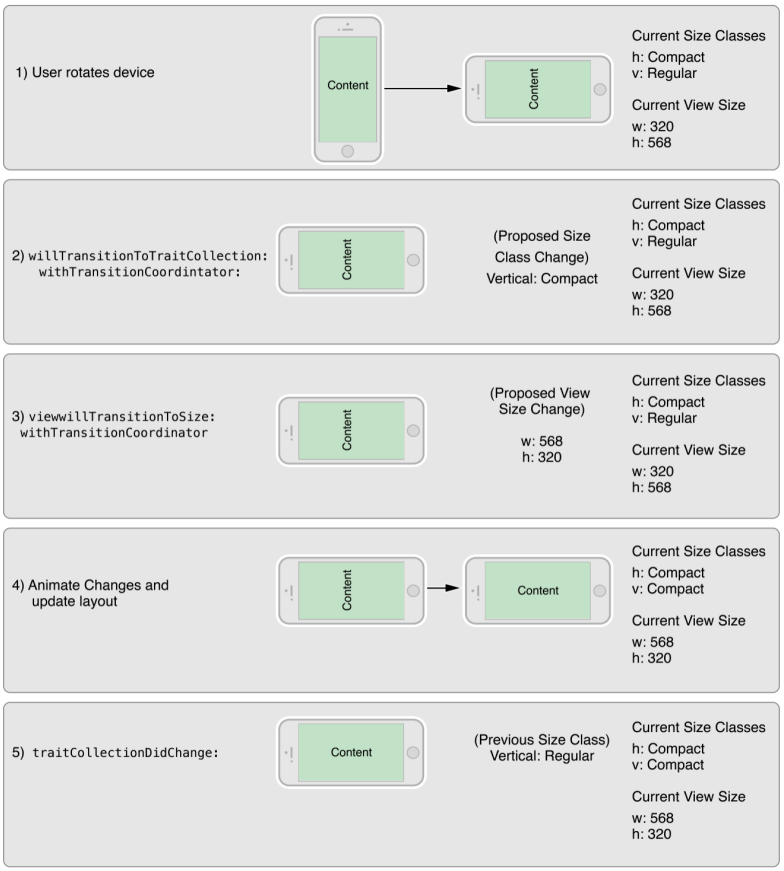
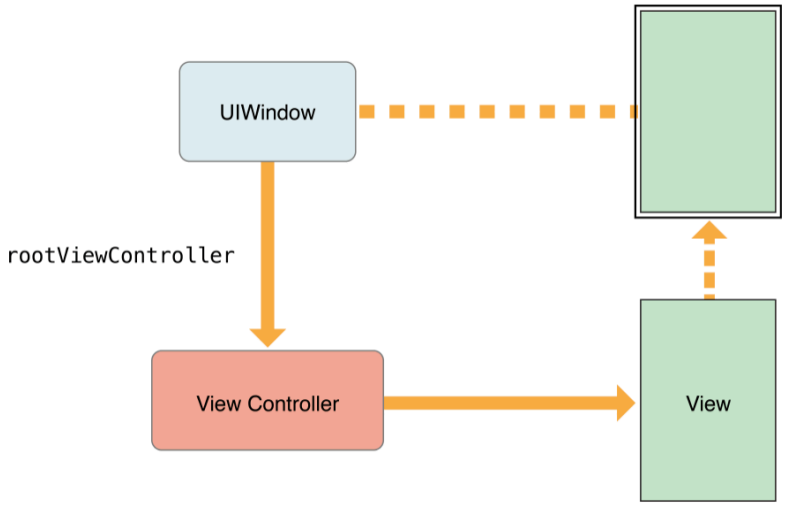


Table 12-2 Size classes for devices with different screen sizes.

Device	Portrait	Landscape
iPad (all) iPad Mini	Vertical size class: Regular Horizontal size class: Regular	Vertical size class: Regular Horizontal size class: Regular
iPhone 6 Plus	Vertical size class: Regular Horizontal size class: Compact	Vertical size class: Compact Horizontal size class: Regular
iPhone 6	Vertical size class: Regular Horizontal size class: Compact	Vertical size class: Compact Horizontal size class: Compact
iPhone 5s iPhone 5c iPhone 5	Vertical size class: Regular Horizontal size class: Compact	Vertical size class: Compact Horizontal size class: Compact
iPhone 4s	Vertical size class: Regular Horizontal size class: Compact	Vertical size class: Compact Horizontal size class: Compact

- The View Controller Hierarchy
  If break prescribed containment & presentation relationships, portions of your app with stop behaving as expect.
    - 1. Root View Controller

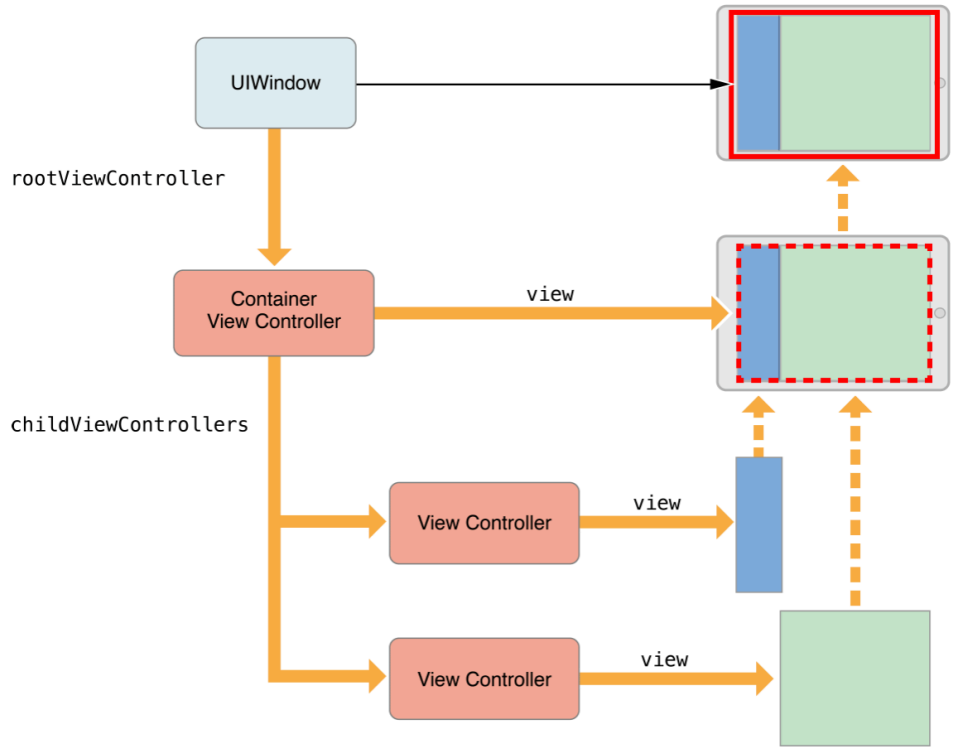
Figure 2-1 The root view controller



- UIWindow.rootViewController

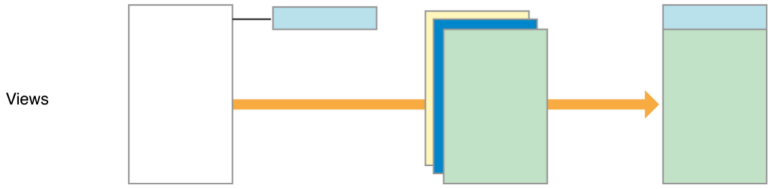
◦ 2. Container View Controller

- `UINavigationController`, `UISplitViewController`, and `UIPageViewController`

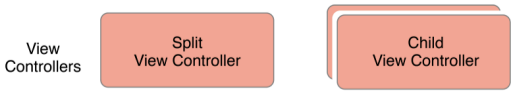


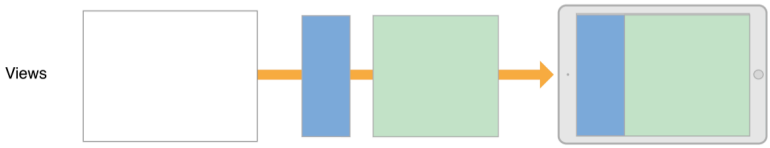
- Implement Custom Container View Controller:
  - example: UINavigationController
    - present one child view controller at a time.

- resize child to fit available space and manage tradition animations.



- example:`UISplitViewController` — master-detail arrangement





- establish parent-child relationship.
  - 1. Call the `addChildViewController:` method of your container view controller. f
  - 2. Add the child's root view to your container's view hierarchy. set the size and position of the child's frame
  - 3. Add any constraints for managing the size and position of the child's root view. Auto layout setup constraints between container and child. The constraints should affect size and position of only child's root view controller.
  - 4. Call the `didMoveToParentViewController:` method of the child view controller.

Listing 5-1 Adding a child view controller to a container

```
- (void) displayContentController: (UIViewController*) content {
    [self addChildViewController:content];
    content.view.frame = [self frameForContentController];
    [self.view addSubview:self.currentClientView];
    [content didMoveToParentViewController:self];
}
```

- Removing a child view controller

Listing 5-2 Removing a child view controller from a container

```
- (void) hideContentController: (UIViewController*) content {
    [content willMoveToParentViewController:nil];
    [content.view removeFromSuperview];
    [content removeFromParentViewController];
}
```

- Transitioning Between Child View Controllers

Listing 5-3 Transitioning between two child view controllers

```
- (void)cycleFromViewController: (UIViewController*) oldVC
                toViewController: (UIViewController*) newVC {
   // Prepare the two view controllers for the change.
   [oldVC willMoveToParentViewController:nil];
   [self addChildViewController:newVC];

   // Get the start frame of the new view controller and the end frame
   // for the old view controller. Both rectangles are offscreen.
   newVC.view.frame = [self newViewStartFrame];
   CGRect endFrame = [self oldViewEndFrame];

   // Queue up the transition animation.
   [self transitionFromViewController: oldVC toViewController: newVC
        duration: 0.25 options:0
        animations:^{
            // Animate the views to their final positions.
            newVC.view.frame = oldVC.view.frame;
            oldVC.view.frame = endFrame;
        }
        completion:^(BOOL finished) {
           // Remove the old view controller and send the final
           // notification to the new view controller.
           [oldVC removeFromParentViewController];
           [newVC didMoveToParentViewController:self];
        }];
}
```

- Managing Appearance updates for children. (Consolidate changes children updating view at same time)
    - first disable automatic appearance forwarding. (override since default is yes)

```
- (BOOL) shouldAutomaticallyForwardAppearanceMethods {
    return NO;
}
```

- Tell child when its views are going to appear or disappear.

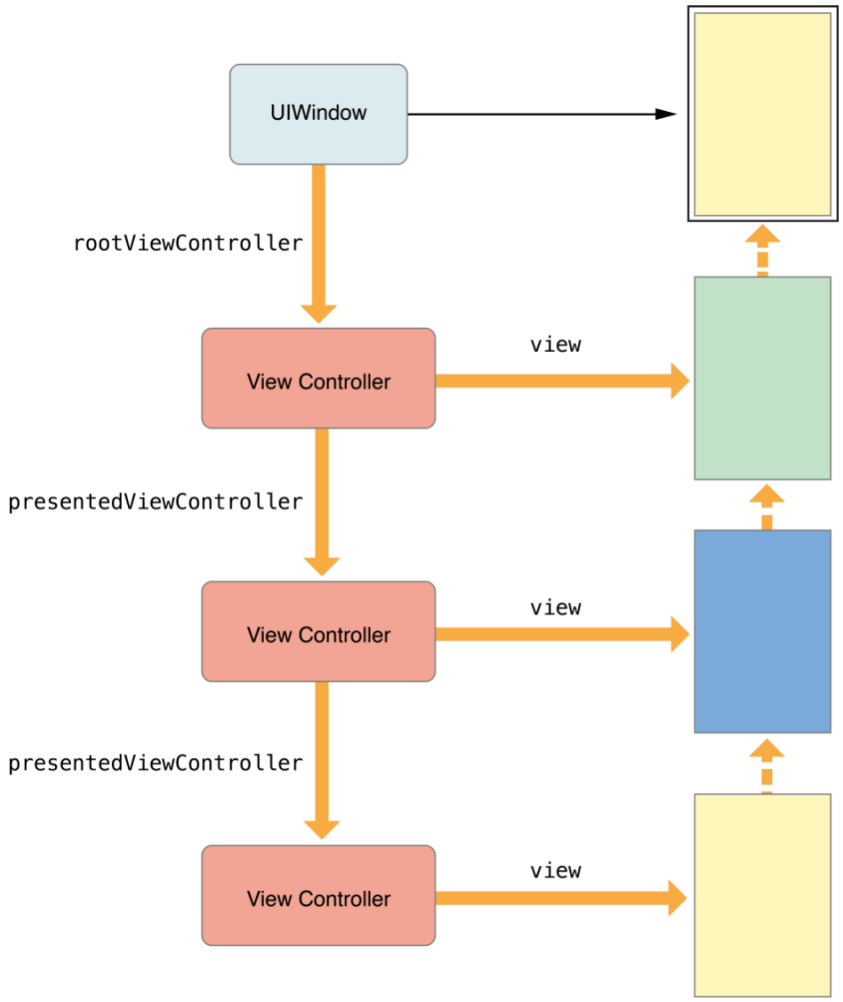
Listing 5-5 Forwarding appearance messages when the container appears or disappears

```
-(void) viewWillAppear:(BOOL)animated {
    [self.child beginAppearanceTransition: YES animated: animated];
}

-(void) viewDidAppear:(BOOL)animated {
    [self.child endAppearanceTransition];
}

-(void) viewWillDisappear:(BOOL)animated {
    [self.child beginAppearanceTransition: NO animated: animated];
}

-(void) viewDidDisappear:(BOOL)animated {
    [self.child endAppearanceTransition];
}
```

-
- Suggestions for building container view controller
  - access only root view of a child view controller. (*child.view*)
  - child view controller should have minimal knowledge of their container.(child view controller focus on own content. if container allows its behavior to be influenced by a child, use *delegation* to manage)
  - design container using regular views first.
- Delegating controller to a child view controller.
  - child view controller determine the status bar style. override one or both of the `childViewControllerForStatusBarStyle` and `childViewControllerForStatusBarHidden` methods in your container view controller.
  - child specify its own preferred size. `preferredContentSize`

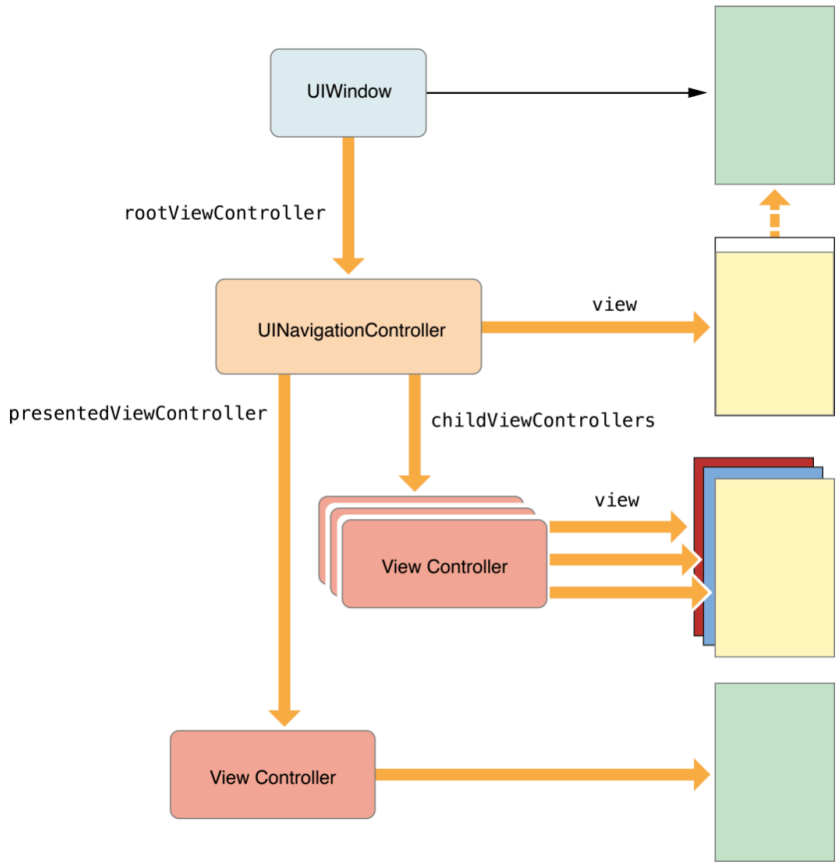
- 3. Presented View Controller (presenting view controller & presented view controller)

Figure 2-3 Presented view controllers



- container view controller: provide context to presentation

Figure 2-4 A container and a presented view controller



    - Presentation and Transition Process (talk about later)
- Design Tips
    - Use System-Supplied View Controllers
        - 1. alertViewController, take pic & video, and managing files on iCloud.
        - 2. GameKit —
        - 3. Address Book UI framework
        - 4. MediaPlayer framework — play&managing video
        - 5. EventKit — displaying & editing user's calendar data
        - 6. GLKit framework — managing an OpenGL rendering surface
        - 7. Multi-peer Connectivity framework — detecting other users and inviting them to connect
        - 8. Message UI framework — composing emails & SMS messages
        - 9. PassKit framework — displaying passes and adding them to Passbook
        - 10. Social framework — composing message for Twitter, FB, other social media sites.
        - 11. AVFoundation framework — displaying media assets
    - Self-contained each View Controller. If needed two view controller pass data, use *delegation*.
      One object defines a *protocol* for communicating with an associated delegate object.

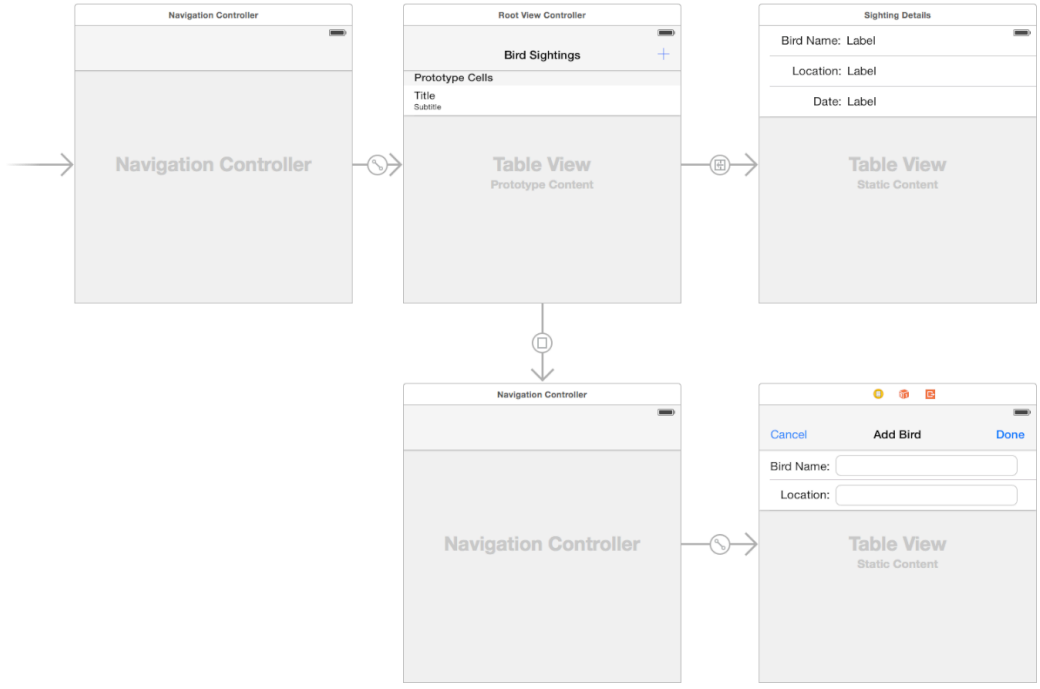
Other objects could conform the protocol.
    - Use Root View only as a container for other views.
    - Know where data lives. Like data validation should in Data object, not controller。
    - Adapt to Changes — use built-in adaptivity support to respond to size and size class changes.

---

2. View Controller Definition

- Defining your subclass.
  (mostly for content view controller:)
  `UITableViewController`, `UICollectionViewController`, and `UIViewController`
  For Container view controller, for existing containers(`UINavigationController`, and `UISplitViewController`) choose vc class you want to modify. for new container view controller, subclass `UIViewController`.
    - Defining UI

Figure 4-1 A storyboard holds a set of view controllers and views



        - Add and configure views
        - connect outlets and actions — chapter 2.1.2 Handling User Interactions
            - view controller define action methods for handling higher-level events.
            - view controller observe *notification* sent by system/other objects.
            - view controller act as a *data source* / *delegate* for another object.

Listing 4-1 Defining outlets and actions in a view controller class

OBJECTIVE-C

```objc
@interface MyViewController : UIViewController
@property (weak, nonatomic) IBOutlet UIButton *myButton;
@property (weak, nonatomic) IBOutlet UITextField *myTextField;

- (IBAction)myButtonAction:(id)sender;

@end
```

SWIFT

```swift
class MyViewController: UIViewController {
    @IBOutlet weak var myButton : UIButton!
    @IBOutlet weak var myTextField : UITextField!

    @IBAction func myButtonAction(sender: id)
}
```

        - create relationships & segues — chapter 3.2 use segue
        - customize layout and views for diff size classes — chapter 4.2 Building an adaptive interface
        - add gesture recognizers to handle user interactions with views.
    - Handling User Interactions — connect outlets and actions, see above section.
    - Displaying your views at runtime (Related topic: *UIViewController Class Reference*)
        - 1. instantiates views
        - 2. connect all outlets and actions
        - 3. assign root view to view controller's view property
        - 4. call `awakeFromNib` mehtod.
        - 5. call `viewDidLoad`.
        - Before displaying:
            - 1. `viewWillAppear:` : method.
            - 2. updates layout of the views.
            - 3. Displays views onscreen
            - 4. call `viewDidAppear:`:
    - Managing view layout (During layout process)
        - 1. update trait collections of view controller and its views (When Do Trait and Size Changes Happen?)
        - 2. call the view controller's `viewWillLayoutSubviews` method.
        - 3. call `containerViewWillLayoutSubviews` method of the current `UIPresentationController` object
        - 4. call `layoutSubviews` method of view controller's root view
        - 5. applies the computed layout info on views.

        - 6. call view controller's `viewDidLayoutSubviews` method.
        - 7. call `containerViewDidLayoutSubviews` method of the current `UIPresentationController` object.
        - Tips:
            - auto layout
            - update constraints when adding/removing views.
            - remove constrains temporarily while animating view controller's view.
        - Related Topic: The Presentation and Transition Process.
    - Managing Memory Efficiently

Table 4-1 Places to allocate and deallocate memory

Task	Methods	Discussion
Allocate critical data structures required by your view controller.	Initialization methods	Your custom initialization method (whether it is named `init` or something else) is always responsible for putting your view controller object into a known good state. Use these methods to allocate whatever data structures are needed to ensure proper operation.
Allocate or load data to be displayed in your view.	`viewDidLoad`	Use the `viewDidLoad` method to load any data objects you intend to display. By the time this method is called, your view objects are guaranteed to exist and to be in a known good state.
Respond to low-memory notifications.	`didReceiveMemoryWarning`	Use this method to deallocate all noncritical objects associated with your view controller. Deallocate as much memory as you can.
Release critical data structures required by your view controller.	`dealloc`	Override this method only to perform any last-minute cleanup of your view controller class. The system automatically releases objects stored in instance variables and properties of your class, so you do not need to release those explicitly.

- Implementing a container view controller (already talked above section)
- Supporting accessibility
    - voice over.
    - special voice over gesture, and observing accessibility notifications.
    - moving voice over cursor to specific element (default from left to right and top to bottom.)
        - post a `UIAccessibilityScreenChangedNotification` notification using

the UIAccessibilityPostNotification function

Listing 6-1 Posting an accessibility notification can change the first element read aloud

```
@implementation MyViewController
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];

    // The second parameter is the new focus element.
    UIAccessibilityPostNotification(UIAccessibilityScreenChangedNotification,
                                    self.myFirstElement);
}
@end
```

- Responding to special voice over Gesture
  - 1. escape
    `func accessibilityPerformEscape() -> Bool`
  - 2. magic tap
    `func accessibilityPerformMagicTap() -> Bool`
  - 3. three finger scroll
    `func accessibilityScroll(_ direction: UIAccessibilityScrollDirection) -> Bool`
  - 4. increment
    `func accessibilityIncrement()`
  - 5. decrement
    `func accessibilityDecrement()`
- Observing Accessibility Notifications
  - UIAccessibilityAnnouncementDidFinishNotification notification

Listing 6-2 Registering as an observer for accessibility notifications

```
@implementation MyViewController
- (void)viewDidLoad
{
    [super viewDidLoad];

    [[NSNotificationCenter defaultCenter]
        addObserver:self
           selector:@selector(didFinishAnnouncement:)
               name:UIAccessibilityAnnouncementDidFinishNotification
             object:nil];
}

- (void)didFinishAnnouncement:(NSNotification *)dict
{
    NSString *valueSpoken = [[dict userInfo]
  objectForKey:UIAccessibilityAnnouncementKeyStringValue];
    NSString *wasSuccessful = [[dict userInfo]
  objectForKey:UIAccessibilityAnnouncementKeyWasSuccessful];
    // ...
}
@end
```

-
- `UIAccessibilityVoiceOverStatusChanged` notification.
- For a list of accessibility notifications you can observe, see *UIAccessibility Protocol Reference*.

- Preserving and Restoring State
  - steps for preserving app's view controller.
    - (require) assign restoration identifier. see following tagging vc for preservation.
    - (require) tell iOS how to create/locate new view controller objects. see restoring vc at launch time.
    - optional, store specific configuration data to return that view controller. see encoding and decoding ...
  - Tagging View Controllers for Preservation
    - `var restorationIdentifier: String? { get set }`
      during subsequent launches, UIKit ask app for help in recreating view controller that were installed the last time your app ran. All parent view controllers must also have restoration identifier.(restoration path from root view controller)
    - Choosing effective restoration identifiers
      - restoration identifier just class name of view controller, but if use

same class many places, need meaningful name.

- excluding groups of view controller

Figure 7-1 Excluding view controllers from the automatic preservation process

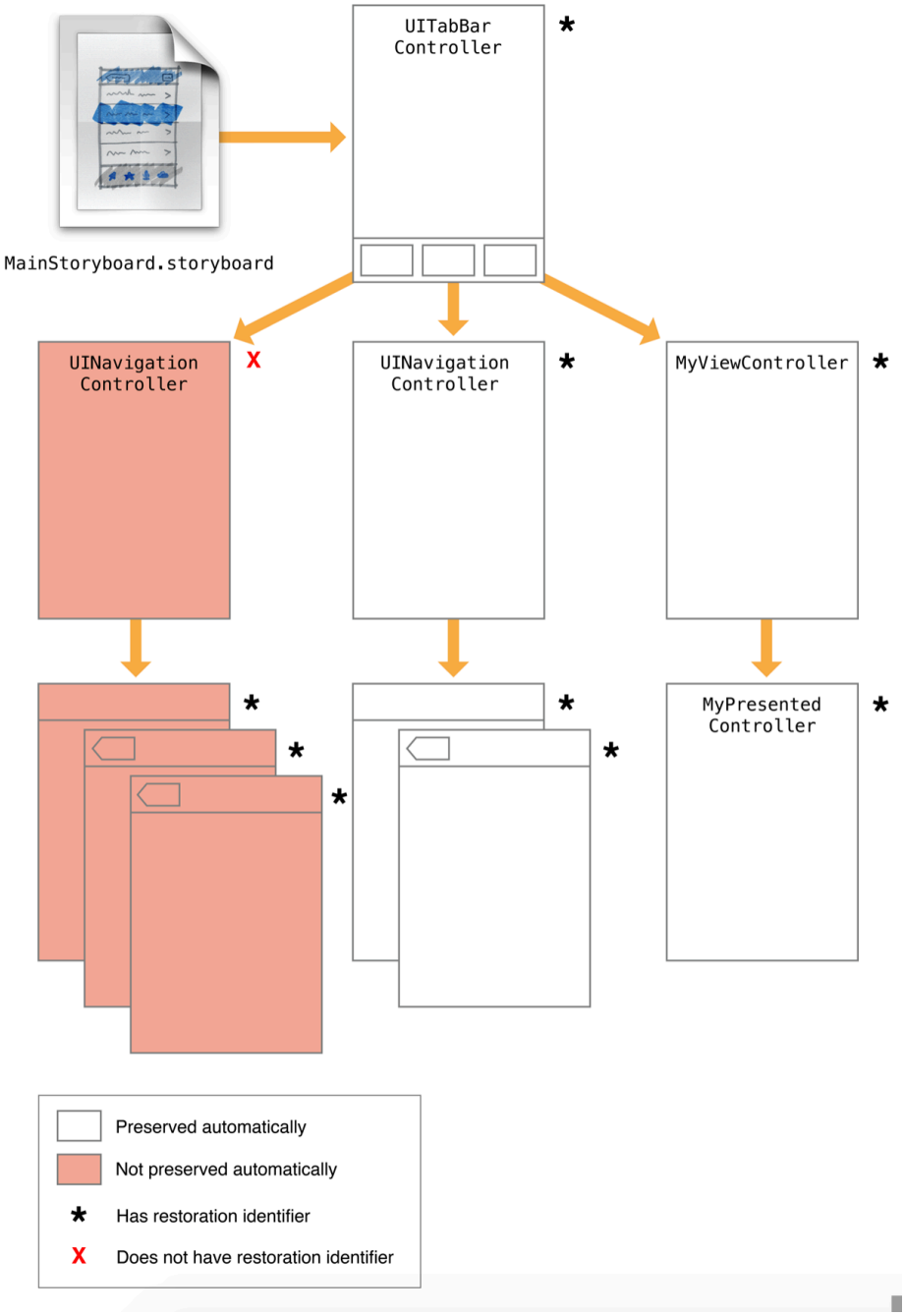
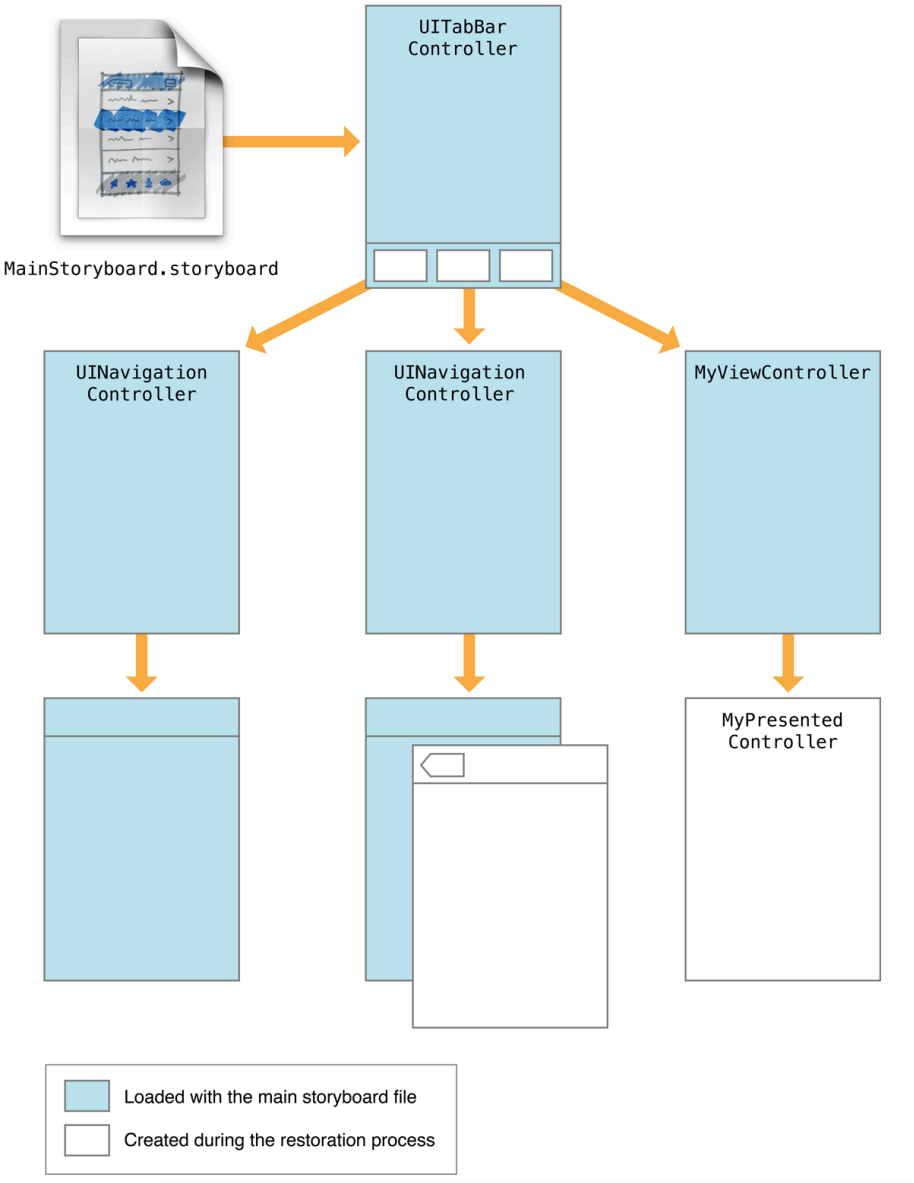


Figure 7-2 Loading the default set of view controllers



- preserving a view controller's view.
  - e.g.: UIScrollView — visual state.
  - assign valid string to view's `restorationIdentifier` t.
  - view's View controller that also has valid restoration identifier.
  - For table view & collection view, assign data source adopts the `UIDataSourceModelAssociation` protocol.

- Restoring View Controllers at Launch Time
  - UIKit search in following order when trying to locate view controller.
  - 1. **If the view controller had a restoration class, UIKit asks that class to provide the view controller.**UIKit calls the `viewControllerWithRestorationIdentifierPath:coder:` method of

the associated restoration class to retrieve the view controller. if that method returns nil, it is assumed that the app does not want to recreate the view controller and UIKit stops looking for it.
`+ (UIViewController *)viewControllerWithRestorationIdentifierPath:(NSArray *)identifierComponents coder:(NSCoder *)coder;`

- 2. **If the view controller did not have a restoration class, UIKit asks the app delegate to provide the view controller.** UIKit calls the application:viewControllerWithRestorationIdentifierPath:coder:method of your app delegate to look for view controllers without a restoration class. If that method returns nil, UIKit tries to find the view controller implicitly.
- 3. **If a view controller with the correct restoration path already exists, UIKit uses that object.** If your app creates view controllers at launch time (either programmatically or by loading them from a storyboard) and those view controllers have restoration identifiers, UIKit finds them implicitly based on their restoration paths.
- 4. **If the view controller was originally loaded from a storyboard file, UIKit uses the saved storyboard information to locate and create it.** loaded from a storyboard, this method uses the UIStateRestorationViewControllerStoryboardKey key to get the storyboard from the archive.

Listing 7-1 Creating a new view controller during restoration

```
+ (UIViewController*) viewControllerWithRestorationIdentifierPath:(NSArray
  *)identifierComponents
                      coder:(NSCoder *)coder {
   MyViewController* vc;
   UIStoryboard* sb = [coder
  decodeObjectForKey:UIStateRestorationViewControllerStoryboardKey];
   if (sb) {
      vc = (PushViewController*)[sb
  instantiateViewControllerWithIdentifier:@"MyViewController"];
      vc.restorationIdentifier = [identifierComponents lastObject];
      vc.restorationClass = [MyViewController class];
   }
    return vc;
}
```

- Encoding and Decoding Your View Controller's State (optional)
  - references to any data being displayed.
  - for a container view controller, references to its child view controller
  - information about current selection
  - For view controllers with a user-configurable view, information about the current configuration of that view.
  - object store must adopt *NSCoding* protocol
  - call super.

Listing 7-2 Encoding and decoding a view controller's state.

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder {
    [super encodeRestorableStateWithCoder:coder];

    [coder encodeInt:self.number forKey:MyViewControllerNumber];
}

- (void)decodeRestorableStateWithCoder:(NSCoder *)coder {
    [super decodeRestorableStateWithCoder:coder];

    self.number = [coder decodeIntForKey:MyViewControllerNumber];
}
```

- Tips for Saving and Restoring your View Controllers
  - might not preserve all view controllers. (displaying a change page,)
  - avoid swapping view controller classes during restoration process.
  - state preservation system expects you to use view controller as they were intended.

3.Presentations and Transitions

- Presenting a View Controller
  Two ways display view controller on screen:
  1. embed it in container view controller,
  2. present it.
  Presenting a view controller creates a relationship between the original view controller, known as the *presenting view controller*, and the new view controller to be displayed, known as the *presented view controller*.
  - Presentation and transition process
    - a. presentation styles
      `var modalPresentationStyle: UIModalPresentationStyle { get set }`
      horizontal compact — modal view controller present as full-screen

`case fullScreen`

A presentation style in which the presented view covers the screen.

`case pageSheet`

A presentation style that partially covers the underlying content.

`case formSheet`

A presentation style that displays the content centered in the screen.

`case currentContext`

A presentation style where the content is displayed over another view controller's content.

`case custom`

A custom view presentation style that is managed by a custom presentation controller and one or more custom animator objects.

`case overFullScreen`

A view presentation style in which the presented view covers the screen.

`case overCurrentContext`

A presentation style where the content is displayed over another view controller's content.

`case blurOverFullScreen`

A presentation style that blurs the underlying content before displaying new content in a full-screen presentation.

Beta

`case popover`

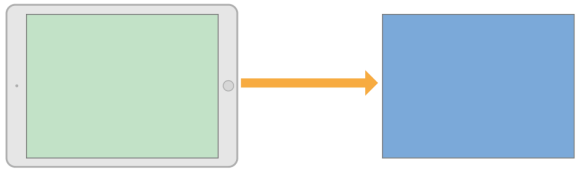
A presentation style where the content is displayed in a popover view.

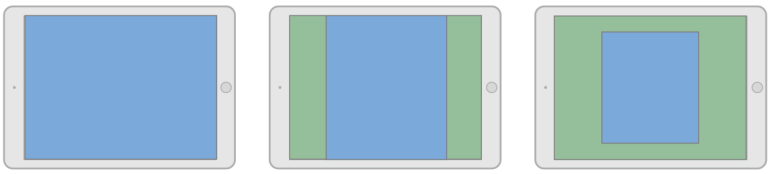
`case none`

A presentation style that indicates no adaptations should be made.

- 1. full-screen presentation styles
  - cover underlying content completely.

Figure 8-1 The full screen presentation styles





  - In Modal Presentation Full-Screen style, remove views underlying view controller after transition animations finish. But if you need those views, could use `UIModalPresentationOverFullScreen` style instead
  - UIKit walks up the view controller hierarchy until finds one could cover entire screen.
- 2. popover style(UIModalPresentationPopover style)

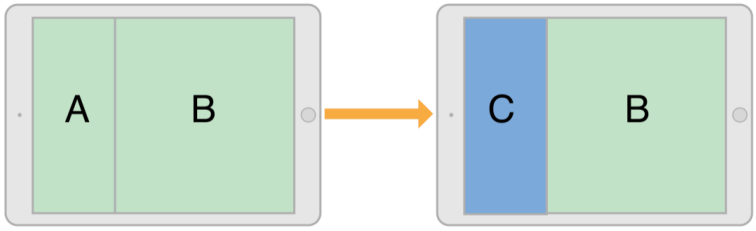
Figure 8-2 The popover presentation style

  - horizontally regular environment, looks like above.

horizontally compact environment, popovers adapt to `UIModalPresentationOverFullScreen` presentation style. In this case, need a way to dismiss a presented popover, like adding a button embedding popover.

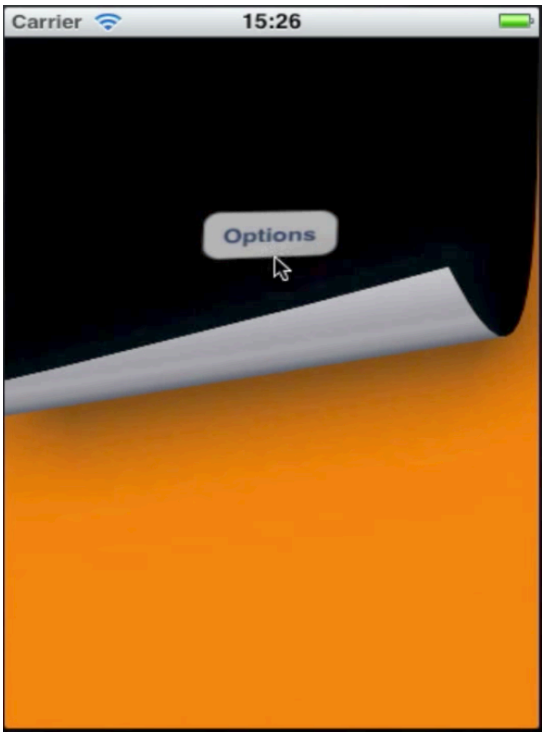
- 3. current context styles

Figure 8-3 The current context presentation style



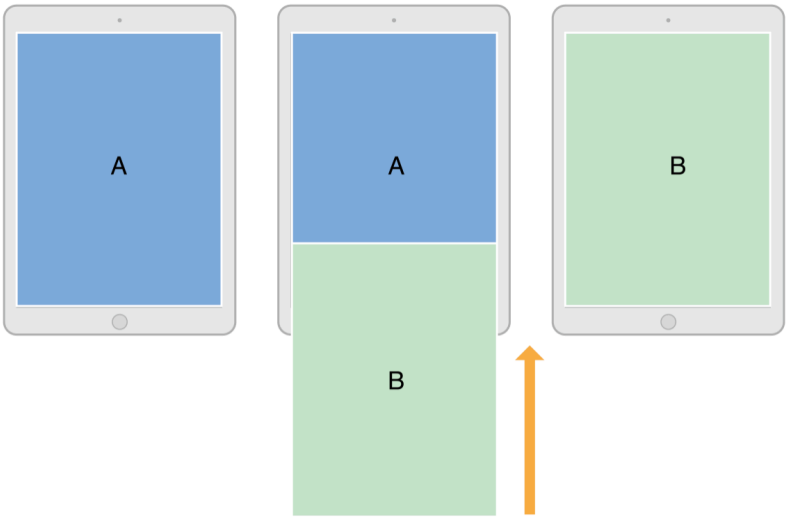
  - also define transition animations to use during presentation.
  - Normally, UIKit animates view controller using value in `modalTransitionStyle`
    `var modalTransitionStyle: UIModalTransitionStyle { get set }`
  - If the presentation context view controller has its `providesPresentationContextTransitionStyle` set to `YES`, UIKit uses the value in that view controller's `modalTransitionStyle` property instead. Default is No, use the transition style of the presented view controller.
  - If horizontal compact, current context style adapt to .FullScreen style. To change the behavior, use an adaptive presentation delegate to specify a different presentation style / view controller.
- 4. custom presentation styles (UIModalPresentationCustom style)
  - creating custom style involves subclassing UIPresentationController and using its methods to animate custom view onto screen, and to set size and position of the presented view controller.
  - see Creating Custom Presentations.

- b. transition styles
  determine type of animations used to display a presented view controller.
  - modalTransitionStyle
    `var modalTransitionStyle: UIModalTransitionStyle { get set }`
    using `present(_:animated:completion:)` method.
    - `coverVertical (default)`
    - `flipHorizontal`
    - `crossDissolve – fade-out and fade-in`
    - `partialCurl`



- `UIModalTransitionStyleCoverVertical` is default

Figure 8-4 A transition animation for a view controller



- c. Presenting v.s. Showing a view controller
  The `UIViewController` class offers two ways to display a view controller:
  - 1. showViewController: sender: and showDetailViewController:sender: methods offer the most adaptive and flexible way to display view controllers. Let presenting view controller decide how best to handle the presentation. In container view controller, show it as child not present it modal.
  - 2. presentViewController: animated: completion: method always displays

the view controller modally. without knowing any about view controller hierarchy.

- Presenting a view controller
  1. Use segue to present view controller automatically.
  2. Use `showViewController:sender:` or `showDetailViewController:sender:`method
  3. Call the presentViewController:animated:completion: method
     - show view controllers (Use `showViewController:sender:` or `showDetailViewController:sender:`)
       - 1. create vc.
       - 2. set `modalPresentationStyle` property might not final presentation
       - 3. Set the modalTransitionStyle property might not final animation
       - 4. Call the showViewController:sender: and showDetailViewController:sender: method of the current view controller
       - Note: UIKit forwards calls to appropriate presenting view controller
     - presenting view controllers modally
       - 1. create vc
       - 2. set of `modalPresentationStyle` new view controller to desired presentation style.
       - 3. set modalTransitionStyle property of the view controller to the desired animation style.
       - 4. call the `presentViewController:animated:completion:` method of the current view controller.

Listing 8-1 Presenting a view controller programmatically

```
- (void)add:(id)sender {
    // Create the root view controller for the navigation controller
    // The new view controller configures a Cancel and Done button for the
    // navigation bar.
    RecipeAddViewController *addController = [[RecipeAddViewController alloc] init];

    addController.modalPresentationStyle = UIModalPresentationFullScreen;
    addController.transitionStyle = UIModalTransitionStyleCoverVertical;
    [self presentViewController:addController animated:YES completion: nil];
}
```

     - presenting view controller in popover
       - popover require additional configuration.
       - 1. Set the `preferredContentSize` property of your view controller to the desired size.
       - 2. Set the popover anchor point using the associated UIPopoverPresentationController object, which is accessible from the view controller's popoverPresentationController property. Set only one of the following:

        - Set the barButtonItem property to a bar button item.
        - Set the sourceView and sourceRect properties to a specific region in one of your views.
    - Dismissing a presented view controller.
        - call the `dismissViewControllerAnimated:completion:` method of the presenting view controller.
        - can also call this method on the presented view controller itself. UIKit automatically forwards the request to the presenting view controller.
    - Presenting view controller defined in a different storyboard

Listing 8-2 Loading a view controller from a storyboard

```
UIStoryboard* sb = [UIStoryboard storyboardWithName:@"SecondStoryboard" bundle:nil];
MyViewController* myVC = [sb
  instantiateViewControllerWithIdentifier:@"MyViewController"];

// Configure the view controller.

// Display the view controller
[self presentViewController:myVC animated:YES completion:nil];
```

- Using Segues

Table 9-1 Adaptive segue types

Segue type	Behavior
Show (Push)	This segue displays the new content using the `showViewController:sender:` method of the target view controller. For most view controllers, this segue presents the new content modally over the source view controller. Some view controllers specifically override the method and use it to implement different behaviors. For example, a navigation controller pushes the new view controller onto its navigation stack. UIKit uses the `targetViewControllerForAction:sender:` method to locate the source view controller.
Show Detail (Replace)	This segue displays the new content using the `showDetailViewController:sender:` method of the target view controller. This segue is relevant only for view controllers embedded inside a `UISplitViewController` object. With this segue, a split view controller replaces its second child view controller (the detail controller) with the new content. Most other view controllers present the new content modally. UIKit uses the `targetViewControllerForAction:sender:` method to locate the source view controller.
Present Modally	This segue displays the view controller modally using the specified presentation and transition styles. The view controller that defines the appropriate presentation context handles the actual presentation.
Present as Popover	In a horizontally regular environment, the view controller appears in a popover. In a horizontally compact environment, the view controller is displayed using a full-screen modal presentation.

- Initiating a segue programmatically
  - 1. The view controller to be presented is created and initialized.
  - 2. The segue object is created and its `initWithIdentifier:source:destination:` method is called. The identifier is the unique string you provided for the segue in Interface Builder, and the two other parameters represent the two controller objects in the transition.
  - 3. The presenting view controller's `prepareForSegue:sender:` method is called. See Modifying a Segue's Behavior at Runtime.
  - 4. The segue object's `perform` method is called. This method performs a transition to bring the new view controller onscreen.

        - 5. The reference to the segue object is released.
- Customizing Transition Animations
    - Transition Animation Sequence
      two types of transitions: 1. presentation, 2. dismissal.
        - Transition Delegate — iOS 7.0
            - define and that conforms to
              the `UIViewControllerTransitioningDelegate` protocol
                - Animator Objects — reveal or hide a view controller's view.
                  All transitions use a *transition animator object*—an object that conforms to
                  the `UIViewControllerAnimatedTransitioning` protocol

                    - `optional func animationController(forPresented presented: UIViewController, presenting: UIViewController, source: UIViewController) -> UIViewControllerAnimatedTransitioning?`
                    - `optional func animationController(forDismissed dismissed: UIViewController) -> UIViewControllerAnimatedTransitioning?`

                - Interactive Animator Objects — drives timeing of custom animations using touch events / gesture recognizer. conform to the UIViewControllerInteractiveTransitioning protocol.

                    - The easiest way to create an interactive animator is to subclass `UIPercentDrivenInteractiveTransition` class and add event-handling code to your subclass.

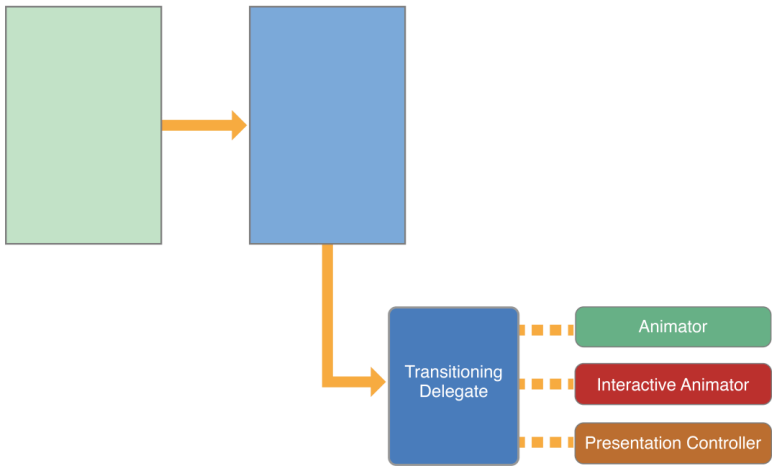
                    - `optional func interactionControllerForPresentation(using animator: UIViewControllerAnimatedTransitioning) -> UIViewControllerInteractiveTransitioning?`
                      `// Here the animator returned by above function animationController(forPresented presented: UIViewController, presenting: UIViewController, source: UIViewController). So, if implement this must implement animationController(forPresented … )`
                    - `optional func interactionControllerForDismissal(using animator: UIViewControllerAnimatedTransitioning) -> UIViewControllerInteractiveTransitioning?`
                      `// If you implement this method, you must also implement`

```
the animationController(forDismissed:) method and
use it to return a custom transition animator
object. If
the animationController(forDismissed:) method
returns nil, UIKit does not call this method.
```

- Presentation Controller
  - manages presentation style while view controller is on-screen.
  - `optional func presentationController(forPresented presented: UIViewController, presenting: UIViewController?, source: UIViewController) -> UIPresentationController?`

- ModalPresentationStyle = .custom, then conform transitioning Delegate. If no provide animator objects, UIKit use standard transition animation in view controller's modalTranstionStyle.

Figure 10-1 The custom presentation and animator objects



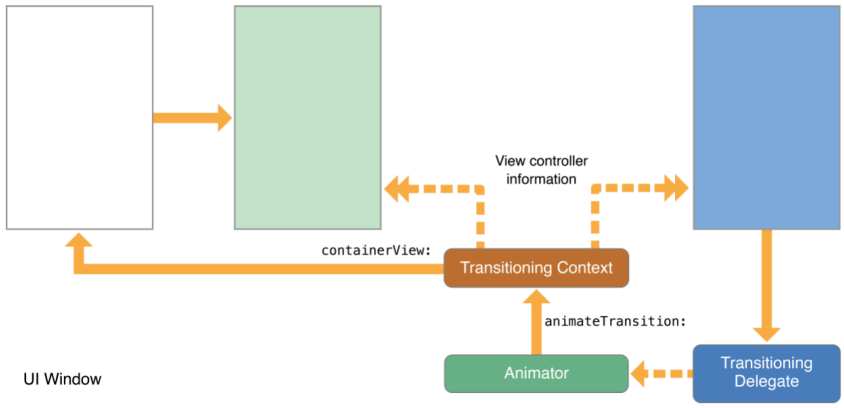
- Custom Animation Sequence
  - set `transitioningDelegate`
  - As it prepares a presentation, UIKit calls the `animationControllerForPresentedController:presentingController:sourceController:` method of your transitioning delegate to retrieve the custom animator object. If an object is available, UIKit performs the following steps:
  - 1. UIKit calls the transitioning delegate's `interactionControllerForPresentation:` method to

see if an interactive animator object is available. If that method returns `nil`, UIKit performs the animations without user interactions.

- 2. UIKit calls the `transitionDuration:` method of the animator object to get the animation duration.
- 3. UIKit calls the appropriate method to start the animations:
  - For non-interactive animations, UIKit calls the `animateTransition:` method of the animator object.
  - For interactive animations, UIKit calls the `startInteractiveTransition:` method of the interactive animator object.
- 4. UIKit waits for an animator object to call the `completeTransition:` method of the context transitioning object.

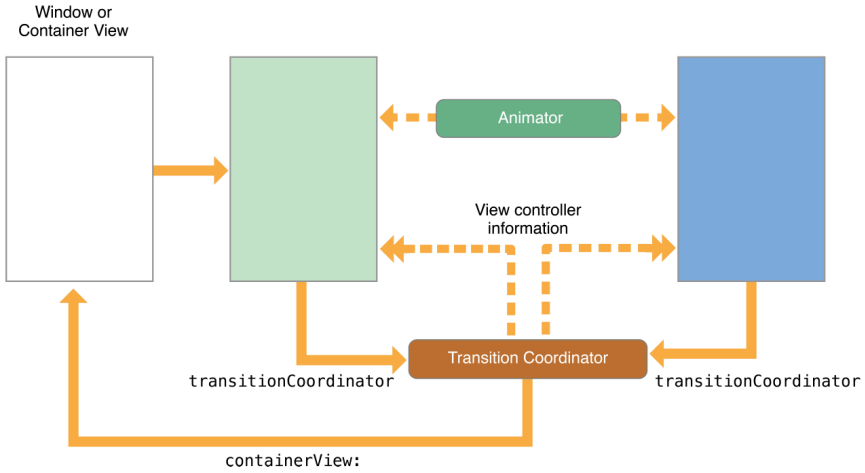
- Transitioning Context Object
  - implements the `UIViewControllerContextTransitioning` protocol
  - use object and data in transitioning context object rather than cached information by your self.

Figure 10-2 The transitioning context object



- Transition Coordinator
  - A transition coordinator exists only for the duration of the transition.

Figure 10-3 The transition coordinator objects



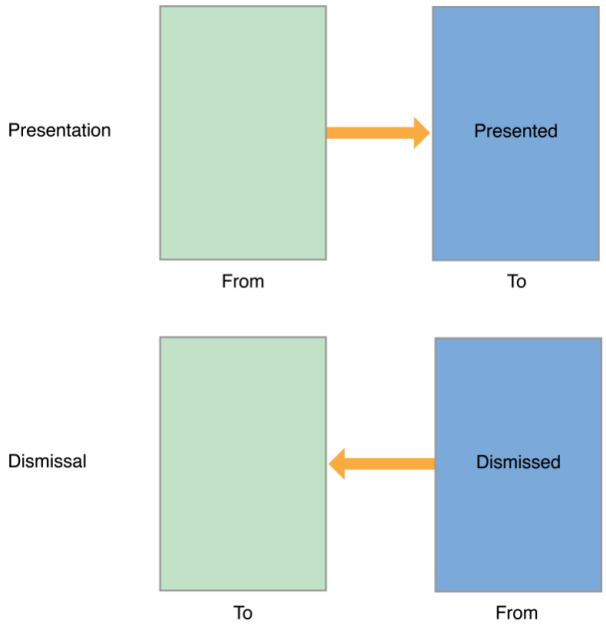
- Presenting a view controller Using Custom Animations
  - 1. Create view controller that want to present.
  - 2. Create custom trasitioning delegate and assign it to view controller's `transitioningDelegate` property. (create and return custom animator objects when asked)
  - 3. Call presentViewController:animated:completion: method to present the view controller.
- Implementing Transitioning Delegate

Listing 10-1 Creating an animator object

```
- (id<UIViewControllerAnimatedTransitioning>)
    animationControllerForPresentedController:(UIViewController *)presented
                         presentingController:(UIViewController *)presenting
                             sourceController:(UIViewController *)source {
    MyAnimator* animator = [[MyAnimator alloc] init];
    return animator;
}
```

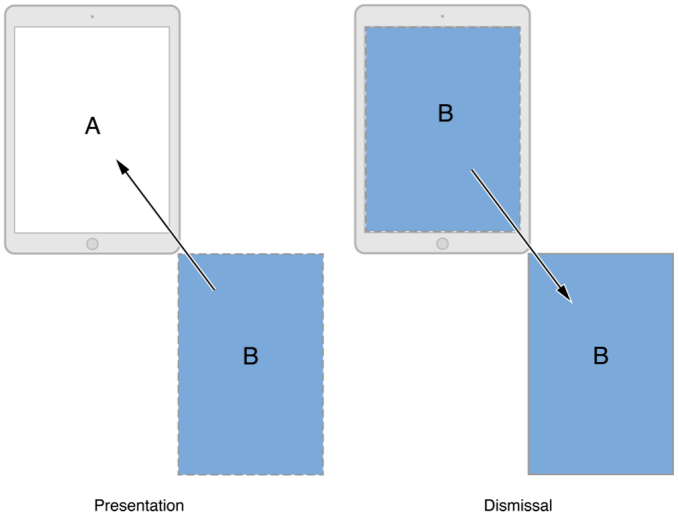
- Implementing Animator Objects
  - Getting Animation Parameters
    - Call the `viewControllerForKey:` method twice to get the "from" and "to" view controller's involved in the transition.
    - Call the `containerView` method to get the superview for the animations.
    - Call the `viewForKey:` method to get the view to be added or removed.
    - Call the `finalFrameForViewController:` method to get the final frame rectangle for the view being added or removed.

Figure 10-4 The from and to objects



- Creating Transition Animations
  - Presentation Animations:
    - Use the `viewControllerForKey:` and `viewForKey:` methods to retrieve the view controllers and views involved in the transition.
    - Set the starting position of the “to” view. Set any other properties to their starting values as well.
    - Get the end position of the “to” view from the `finalFrameForViewController:` method of the context transitioning context.
    - Add the “to” view as a subview of the container view.
    - Create the animations.
  - Dismissal Animations:
    - Similar to presentation animations

Figure 10-5 A custom presentation and dismissal



Listing 10-2 Animations for implementing a diagonal presentation and dismissal

```
- (void)animateTransition:
  (id<UIViewControllerContextTransitioning>)transitionContext {
    // Get the set of relevant objects.
    UIView *containerView = [transitionContext containerView];
    UIViewController *fromVC = [transitionContext
            viewControllerForKey:UITransitionContextFromViewControllerKey];
    UIViewController *toVC   = [transitionContext
            viewControllerForKey:UITransitionContextToViewControllerKey];

    UIView *toView = [transitionContext viewForKey:UITransitionContextToViewKey];
    UIView *fromView = [transitionContext
  viewForKey:UITransitionContextFromViewKey];

    // Set up some variables for the animation.
    CGRect containerFrame = containerView.frame;
    CGRect toViewStartFrame = [transitionContext
  initialFrameForViewController:toVC];
    CGRect toViewFinalFrame = [transitionContext finalFrameForViewController:toVC];
    CGRect fromViewFinalFrame = [transitionContext
  finalFrameForViewController:fromVC];

    // Set up the animation parameters.
    if (self.presenting) {
        // Modify the frame of the presented view so that it starts
        // offscreen at the lower-right corner of the container.
        toViewStartFrame.origin.x = containerFrame.size.width;
        toViewStartFrame.origin.y = containerFrame.size.height;
    }
    else {
        // Modify the frame of the dismissed view so it ends in
        // the lower-right corner of the container view.
        fromViewFinalFrame = CGRectMake(containerFrame.size.width,
                                        containerFrame.size.height,
                                        toView.frame.size.width,
                                        toView.frame.size.height);
    }

```

```
    // Always add the "to" view to the container.
    // And it doesn't hurt to set its start frame.
    [containerView addSubview:toView];
    toView.frame = toViewStartFrame;

    // Animate using the animator's own duration value.
    [UIView animateWithDuration:[self transitionDuration:transitionContext]
                     animations:^{
                         if (self.presenting) {
                             // Move the presented view into position.
                             [toView setFrame:toViewFinalFrame];
                         }
                         else {
                             // Move the dismissed view offscreen.
                             [fromView setFrame:fromViewFinalFrame];
                         }
                     }
                     completion:^(BOOL finished){
                         BOOL success = ![transitionContext transitionWasCancelled];

                         // After a failed presentation or successful dismissal, remove the view.
                         if ((self.presenting && !success) || (!self.presenting && success)) {
                             [toView removeFromSuperview];
                         }

                         // Notify UIKit that the transition has finished
                         [transitionContext completeTransition:success];
                     }];

}
```

- Cleaning up After Animations
  - must call `completeTransition:` method. trigger a cascade of other completion handlers.
    including the one from
    the `presentViewController:animated:completion:` method and the animator object's own `animationEnded:`method. And best place call completeTransition: method is in the completion handler of your animation block.
    `func completeTransition(_ didComplete: Bool)`
  - `transition could be canceled by using return value of transitionWasCancelled method.`
- Adding Interactivity to Transitions
  - The easiest way to make your animations interactive is to use a `UIPercentDrivenInteractiveTransition`object.

Listing 10-3 Configuring a percent-driven interactive animator

```
- (void)startInteractiveTransition:
  (id<UIViewControllerContextTransitioning>)transitionContext {
    // Always call super first.
    [super startInteractiveTransition:transitionContext];

    // Save the transition context for future reference.
    self.contextData = transitionContext;

    // Create a pan gesture recognizer to monitor events.
    self.panGesture = [[UIPanGestureRecognizer alloc]
                        initWithTarget:self action:@selector(handleSwipeUpdate:)];
    self.panGesture.maximumNumberOfTouches = 1;

    // Add the gesture recognizer to the container view.
    UIView* container = [transitionContext containerView];
    [container addGestureRecognizer:self.panGesture];
}
```

Listing 10-4 Using events to update the animation progress

```
-(void)handleSwipeUpdate:(UIGestureRecognizer *)gestureRecognizer {
    UIView* container = [self.contextData containerView];

    if (gestureRecognizer.state == UIGestureRecognizerStateBegan) {
        // Reset the translation value at the beginning of the gesture.
        [self.panGesture setTranslation:CGPointMake(0, 0) inView:container];
    }
    else if (gestureRecognizer.state == UIGestureRecognizerStateChanged) {
        // Get the current translation value.
        CGPoint translation = [self.panGesture translationInView:container];

        // Compute how far the gesture has travelled vertically,
        //  relative to the height of the container view.
        CGFloat percentage = fabs(translation.y /
  CGRectGetHeight(container.bounds));

        // Use the translation value to update the interactive animator.
        [self updateInteractiveTransition:percentage];
    }
    else if (gestureRecognizer.state >= UIGestureRecognizerStateEnded) {
        // Finish the transition and remove the gesture recognizer.
        [self finishInteractiveTransition];
        [[self.contextData containerView] removeGestureRecognizer:self.panGesture];
    }
}
```

- Creating Animations that Run Alongside a Transition —

(`transitionCoordinator`)
        - The transition coordinator exists only while a transition is in progress.
        - To create animations, call the `animateAlongsideTransition:completion:` or `animateAlongsideTransitionInView:animation:completion:` method of the transition coordinator.
          `- (BOOL)animateAlongsideTransition:(void (^)(id<UIViewControllerTransitionCoordinatorContext> context))animation completion:(void (^)(id<UIViewControllerTransitionCoordinatorContext> context))completion;`
          `// YES if the animations were successfully queued to run or NO if they were not.`
            - Use this method to perform animations that are not handled by the animator objects themselves.
        - `func animateAlongsideTransition(in view: UIView?, animation: ((UIViewControllerTransitionCoordinatorContext) -> Void)?, completion: ((UIViewControllerTransitionCoordinatorContext) -> Void)? = nil) -> Bool`
    - Using a Presentation Controller with your Animations
        - custom presentation controller from transitioning delegate of presented view controller
- Creating Custom Presentations
    - Custom Presentation Process
        - presentation controller works alongside any animator objects. Animator objects animate view controller's content onto screen and presentation controller handle everything else.
        - During presentation:
            - 1. Calls the `presentationControllerForPresentedViewController:presentingViewController:sourceViewController:` method of the transitioning delegate to retrieve your custom presentation controller
              `optional func presentationController(forPresented presented: UIViewController, presenting: UIViewController?, source: UIViewController) -> UIPresentationController?`

During a presentation, UIKit:

1. Calls the `presentationControllerForPresentedViewController:presentingViewController:sourceViewController:` method of the transitioning delegate to retrieve your custom presentation controller
2. Asks the transitioning delegate for the animator and interactive animator objects, if any
3. Calls your presentation controller's `presentationTransitionWillBegin` method

   Your implementation of this method should add any custom views to the view hierarchy and configure the animations for those views.
4. Gets the `presentedView` from your presentation controller

   The view returned by this method is animated into position by the animator objects. Normally, this method returns the root view of the presented view controller. Your presentation controller can replace that view with a custom background view, as needed. If you do specify a different view, you must embed the root view of the presented view controller into your view hierarchy.
5. Performs the transition animations

   The transition animations include the main ones created by the animator objects and any animations you configured to run alongside the main animations. For information on the transition animations, see The Transition Animation Sequence.

   During the animation process, UIKit calls the `containerViewWillLayoutSubviews` and `containerViewDidLayoutSubviews` methods of your presentation controller so that you can adjust the layout of your custom views as needed.
6. Calls the `presentationTransitionDidEnd:` method when the transition animations finish

During a dismissal, UIKit:

1. Gets your custom presentation controller from the currently visible view controller
2. Asks the transitioning delegate for the animator and interactive animator objects, if any
3. Calls your presentation controller's `dismissalTransitionWillBegin` method

   Your implementation of this method should add any custom views to the view hierarchy and configure the animations for those views .
4. Gets the already `presentedView` from your presentation controller
5. Performs the transition animations

   The transition animations include the main ones created by the animator objects and any animations you configured to run alongside the main animations. For information on the transition animations, see The Transition Animation Sequence.

   During the animation process, UIKit calls the `containerViewWillLayoutSubviews` and `containerViewDidLayoutSubviews` methods of your presentation controller so that you can remove any custom constraints.
6. Calls the `dismissalTransitionDidEnd:` method when the transition animations finish

- Create custom Presentation Controller
  - Setting frame of Presented View Controller

Listing 11-1 Changing the frame of a presented view controller

```
- (CGRect)frameOfPresentedViewInContainerView {
    CGRect presentedViewFrame = CGRectZero;
    CGRect containerBounds = [[self containerView] bounds];

    presentedViewFrame.size = CGSizeMake(floorf(containerBounds.size.width / 2.0),
                                         containerBounds.size.height);
    presentedViewFrame.origin.x = containerBounds.size.width -
                                  presentedViewFrame.size.width;
    return presentedViewFrame;
}
```

- Managing and Animating Custom Views

Listing 11-2 Initializing the presentation controller

```
- (instancetype)initWithPresentedViewController:(UIViewController
  *)presentedViewController
                    presentingViewController:(UIViewController
  *)presentingViewController {
    self = [super initWithPresentedViewController:presentedViewController
                         presentingViewController:presentingViewController];
    if(self) {
        // Create the dimming view and set its initial appearance.
        self.dimmingView = [[UIView alloc] init];
        [self.dimmingView setBackgroundColor:[UIColor colorWithWhite:0.0
  alpha:0.4]];
        [self.dimmingView setAlpha:0.0];
    }
    return self;
}
```

Listing 11-3 Animating the dimming view onto the screen

```
- (void)presentationTransitionWillBegin {
    // Get critical information about the presentation.
    UIView* containerView = [self containerView];
    UIViewController* presentedViewController = [self presentedViewController];

    // Set the dimming view to the size of the container's
    // bounds, and make it transparent initially.
    [[self dimmingView] setFrame:[containerView bounds]];
    [[self dimmingView] setAlpha:0.0];

    // Insert the dimming view below everything else.
    [containerView insertSubview:[self dimmingView] atIndex:0];

    // Set up the animations for fading in the dimming view.
    if([presentedViewController transitionCoordinator]) {
        [[presentedViewController transitionCoordinator]

  animateAlongsideTransition:^(id<UIViewControllerTransitionCoordinatorContext>
                                          context) {
          // Fade in the dimming view.
          [[self dimmingView] setAlpha:1.0];
        } completion:nil];
    }
    else {
        [[self dimmingView] setAlpha:1.0];
    }
}
```

Listing 11-4 Handling a cancelled presentation

```
- (void)presentationTransitionDidEnd:(BOOL)completed {
    // If the presentation was canceled, remove the dimming view.
    if (!completed)
        [self.dimmingView removeFromSuperview];
}
```

Listing 11-5 Dismissing the presentation's views

```
- (void)dismissalTransitionWillBegin {
    // Fade the dimming view back out.
    if([[self presentedViewController] transitionCoordinator]) {
        [[[self presentedViewController] transitionCoordinator]

  animateAlongsideTransition:^(id<UIViewControllerTransitionCoordinatorContext>
                                        context) {
            [[self dimmingView] setAlpha:0.0];
        } completion:nil];
    }
    else {
        [[self dimmingView] setAlpha:0.0];
    }
}

- (void)dismissalTransitionDidEnd:(BOOL)completed {
    // If the dismissal was successful, remove the dimming view.
    if (completed)
        [self.dimmingView removeFromSuperview];
}
```

- Vending your Presentation Controller to UIKit

Listing 11-6 Creating a custom presentation controller

```
- (UIPresentationController *)presentationControllerForPresentedViewController:
                                    (UIViewController *)presented
        presentingViewController:(UIViewController *)presenting
            sourceViewController:(UIViewController *)source {

    MyPresentationController* myPresentation = [[MyPresentationController]
        initWithPresentedViewController:presented
  presentingViewController:presenting];

    return myPresentation;
}
```

- Adapting to Different Size Classes

4.Adaptivity & Size Changes

- Adaptive Model (talked above section)

- Building an adaptive interface