Views

view objects have many responsibilities:
=> Layout and subview management
A view defines its own default resizing behaviors in relation to its parent view.
A view can manage a list of subviews.
A view can override the size and position of its subviews as needed.
A view can convert points in its coordinate system to the coordinate systems of other views or the window.
=> Drawing and animation
A view draws content in its rectangular area.
Some view properties can be animated to new values.
=> Event handling
A view can receive touch events.
A view participates in the responder chain.

Creating and Configuring View Objects

> Creating View Objects Using Interface Builder
1. The simplest way to create views is to assemble them graphically using interface builder. The view objects created by this way is saving in a nib file.

2. The top level of the nib file usually contains a single view object that represents your view controller's view. The top level view should be sized appropriately for the device and contain all objects that are to be presented.

3. When using nib files with view controller, all you have to do is initialize the view controller with the nib file information. The view controller handles the loading and unloading of your views at the appropriate time.

> Creating View Objects Programmatically
1. If you prefer to create view objects programmatically, you can use the standard allocation/initialization pattern. The default initialization method for views is initWithFrame: method.
Example Code

```
CGRect  viewRect = CGRectMake(0, 0, 100, 100);
UIView* myView = [[UIView alloc] initWithFrame:viewRect];
```

> Setting the Properties of a View

1. Usage of some key view properties.

Properties	Usage
alpha, hidden, opaque	These properties affect the opacity of the view. The alpha and hidden properties change the view's opacity directly. The opaque property tells the system how it should composite your view. Set this property to YES if your view's content is fully opaque and therefore does not reveal any of the underlying view's content. Setting this property to YES improves performance by eliminating unnecessary compositing operations.
bounds, frame, center, transform	These properties affect the size and position of the view. The center and frame properties represent the position of the view relative to its parent view. The frame also includes the size of the view. The bounds property defines the view's visible content area in its own coordinate system. The transform property is used to animate or move the entire view in complex ways. For example, you would use a transform to rotate or scale the view. If the current transform is not the identity transform, the frame property is undefined and should be ignored. For information about the relationship between the bounds, frame, and center properties, see "The Relationship of the Frame, Bounds, and Center Properties." For information about how transforms affect a view, see "Coordinate System Transformations."
autoresizingMask, autoresizesSubviews	These properties affect the automatic resizing behavior of the view and its subviews. The autoresizingMask property controls how a view responds to changes in its parent view's bounds. The autoresizesSubviews property controls whether the current view's subviews are resized at all.

contentMode, contentStretch, contentScaleFactor	These properties affect the rendering behavior of content inside the view. The contentMode and contentStretch properties determine how the content is treated when the view's width or height changes. The contentScaleFactor property is used only when you need to customize the drawing behavior of your view for high-resolution screens. For more information on how the content mode affects your view, see "Content Modes." For information about how the content stretch rectangle affects your view, see "Stretchable Views." For information about how to handle scale factors, see "Supporting High-Resolution Screens" in Drawing and Printing Guide for iOS.
gestureRecognizers, userInteractionEnabled, multipleTouchEnabled, exclusiveTouch	These properties affect how your view processes touch events. The gestureRecognizers property contains gesture recognizers attached to the view. The other properties control what touch events the view supports. For information about how to respond to events in your views, see Event Handling Guide for iOS.
backgroundColor, subviews, drawRect: method, layer, (layerClass method)	These properties and methods help you manage the actual content of your view. For simple views, you can set a background color and add one or more subviews. The subviews property itself contains a read-only list of subviews, but there are several methods for adding and rearranging subviews. For views with custom drawing behavior, you must override the drawRect: method. For more advanced content, you can work directly with the view's Core Animation layer. To specify an entirely different type of layer for the view (such as a layer that supports OpenGL ES drawing calls), you must override the layerClass method.

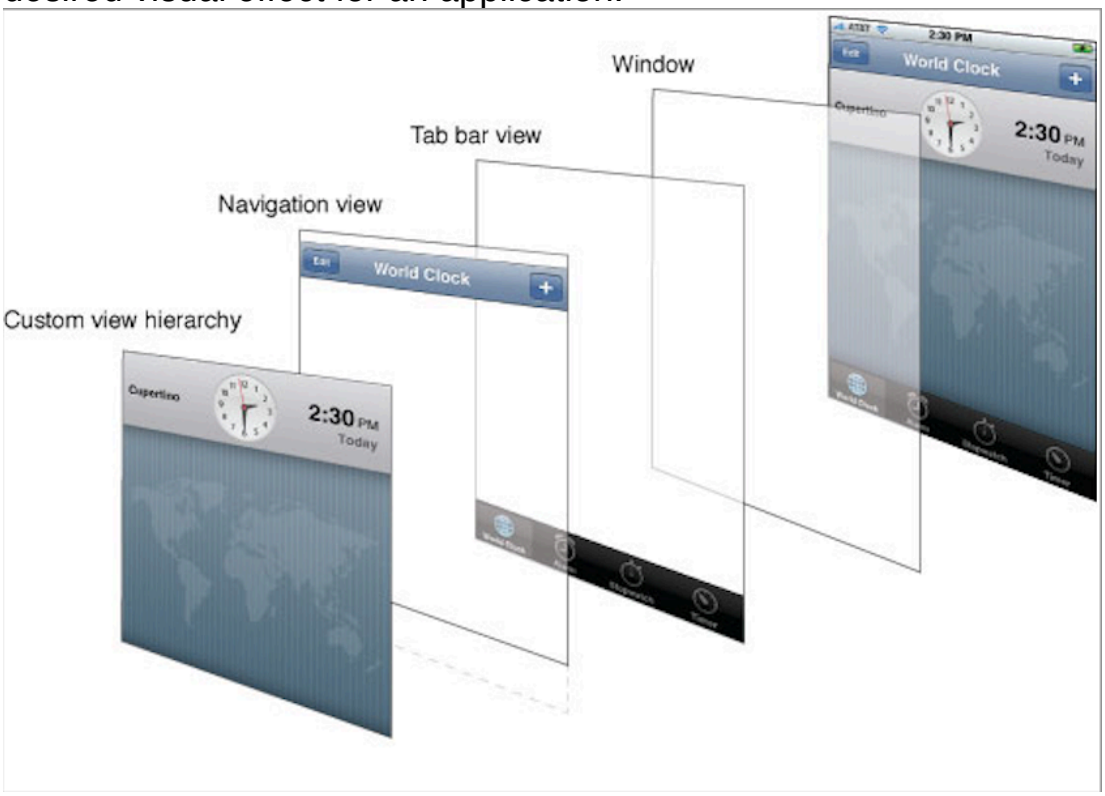
> Tagging Views for Future Identification

1. The UIView class contains a tag property that you can use to tag

individual view objects with integer value. You can use tag to identify views inside your view hierarchy and to perform searches for those views at runtime.

2. To search a tagged view, use the viewWithTag: method, it only search the receiver and its subviews.

Creating and Managing a View Hierarchy
Figure below shows an example of how the layering of views creates the desired visual effect for an application.



> Adding and Removing Subviews
1. Using interface builder to build view hierarchies.

2. If you prefer to create your views programmatically, you create and initialize them and then use the following methods to arrange them into hierarchies:
=> To add a subview to a parent, call the addSubview: method of the parent view. This method adds the subview to the end of the parent's list of subviews.
=> To insert a subview in the middle of the parent's list of subviews, call any of the insertSubview:... methods of the parent view. Inserting a subview in the middle of the list visually places that view behind any views that come later in the list.
=> To reorder existing subviews inside their parent, call the

bringSubviewToFront:, sendSubviewToBack:, or exchangeSubviewAtIndex:withSubviewAtIndex: methods of the parent view. Using these methods is faster than removing the subviews and reinserting them.
=> To remove a subview from its parent, call the removeFromSuperview method of the subview (not the parent view).

3. When adding a subview to its parent, the subview whose frame lies outside of its superview's visible bounds is not clipped by default.

4. The most common example of adding a subview to another view occurs in the application:didFinishLaunchingWithOptions: method of almost every application.
Adding a view to a window

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.

    // Add the view controller's view to the window and display.
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];

    return YES;
}
```

5. Another common place where you might add subviews to a view hierarchy is in the loadView or viewDidLoad methods of a view controller.
Adding views to an existing view hierarchy

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.title = NSLocalizedString(@"TransitionsTitle", @"");

    // create the container view which we will use for transition animation
(centered horizontally)
    CGRect frame = CGRectMake(round((self.view.bounds.size.width -
kImageWidth) / 2.0),
                                        kTopPlacement, kImageWidth,
```

```
kImageHeight);
    self.containerView = [[[UIView alloc] initWithFrame:frame]
autorelease];
    [self.view addSubview:self.containerView];

    // The container view can represent the images for accessibility.
    [self.containerView setIsAccessibilityElement:YES];
    [self.containerView
setAccessibilityLabel:NSLocalizedString(@"ImagesTitle", @"")];

    // create the initial image view
    frame = CGRectMake(0.0, 0.0, kImageWidth, kImageHeight);
    self.mainView = [[[UIImageView alloc] initWithFrame:frame]
autorelease];
    self.mainView.image = [UIImage imageNamed:@"scene1.jpg"];
    [self.containerView addSubview:self.mainView];

    // create the alternate image view (to transition between)
    CGRect imageFrame = CGRectMake(0.0, 0.0, kImageWidth,
kImageHeight);
    self.flipToView = [[[UIImageView alloc] initWithFrame:imageFrame]
autorelease];
    self.flipToView.image = [UIImage imageNamed:@"scene2.jpg"];
}
```

6. When you add subview to another view, UIKit notifies both the parent and child views of the change.

7. After creating a view hierarchy, you can navigate it using the superview and subview properties of your views.

> Hiding Views
To hide a view, you can either use its hidden property to YES or change its alpha property to 0.0.

> Locating Views in a View Hierarchy
Two ways to locate views:
=> Store pointers to any relevant views in an appropriate location, such as in the view controller that owns the views.
=> Assign a unique integer to each view’s tag property and use the viewWithTag: method to locate it.

> Translating, Scaling, and Rotating Views
The transform property of UIView contains a CGAffineTransform structure with the transformations to apply.

> Converting Coordinates in the View Hierarchy
1. The UIView class defines the following methods for converting coordinates to and from the view's local coordinate system.
convertPoint:fromView:
convertRect:fromView:
convertPoint:toView:
convertRect:toView:

2. The UIWindow class also defines the following methods for converting window's coordinates.
convertPoint:fromWindow:
convertRect:fromWindow:
convertPoint:toWindow:
convertRect:toWindow:

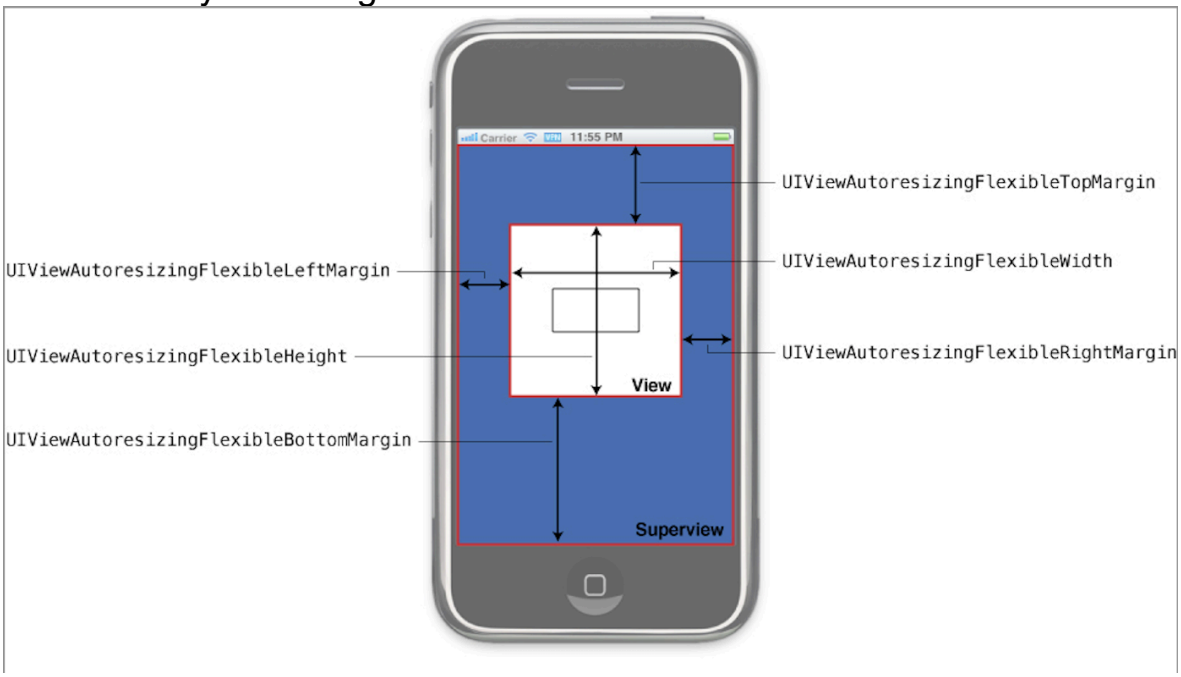
Adjusting the Size and Position of Views at Runtime
With automatic layout, you set the rules that each view should follow when its parent view resizes, and then forget about resizing operation.
With manual layout, you manually adjust the size and position of views as needed.

> Being Prepared for Layout Changes
Layout changes can occur whenever any of the following events happens in a view:
=> The size of a view's bounds rectangle changes.
=> An interface orientation change occurs, which usually triggers a change in the root view's bounds rectangle.
=> The set of Core Animation sublayers associated with the view's layer changes and requires layout.
=> Your application forces layout to occur by calling the setNeedsLayout or layoutIfNeeded method of a view.
=> Your application forces layout by calling the setNeedsLayout method of the view's underlying layer object.

> Handling Layout Changes Automatically Using Autoresizing Rules
1. The autoresizesSubviews property of the superview determines

whether the subview resizes at all.

2. For each view in view hierarchy, setting that view's autoresizingMask property to an appropriate value is an important part of handling automatic layout changes.



> Tweaking the Layout of Your Views Manually

1. You can implement the layoutSubviews method in custom views when the autoresizing behaviors by themselves do not yield the results you want. Your implementation of this method can do the following:
=> Adjust the size and position of any immediate subviews.
=> Add or remove subviews or Core Animation layers.
=> Force a subview to be redrawn by calling its setNeedsDisplay or setNeedsDisplayInRect: method.

2. One place where your application often lay out subdues manually is when implementing a large scrollable area.

3. When writing your layout code, be sure to rest your code in these two ways:
=> Change the orientation of your views to make sure the layout looks correct in all supported interface orientations.
=> Make sure your code responds appropriately to changes in the height of the status bar. When a phone call is active, the status bar height increases in size, and when the user ends the call, the status bar decreases in size.

Modifying Views at Runtime
An application might modify its views by rearranging them, changing their size or position, hiding or showing them, or loading an entirely new set of views. In iOS applications, there are several places and ways in which you perform these kinds of actions:
=> In a view controller
=> In animation blocks
=> Other ways:

Interacting with Core Animation Layers
The layer object for the view is stored in the view's layer property.

> Changing the Layer Class Associated with a View
1. The type of layer associated with a view cannot be changed after the view is created. Therefore, each view uses the layerClass class method to specify the class of its layer object.

2. The view always assign itself as the delegate of its layer object.. So the view owns the layer object, and the relationship between the view and layer object must not change.

> Embedding Layer Objects in a View
1. A custom layer object is an instance of CALayer that is not owned by a view. Custom layers do not receive events or participate in the responder chain but drew themselves and respond to size changes in their parent view or layer accruing to Core Animation rules.
Adding a custom layer to a view

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // Create the layer.
    CALayer* myLayer = [[CALayer alloc] init];

    // Set the contents of the layer to a fixed image. And set
    // the size of the layer to match the image size.
    UIImage layerContents = [[UIImage imageNamed:@"myImage"]
retain];
    CGSize imageSize = layerContents.size;

    myLayer.bounds = CGRectMake(0, 0, imageSize.width,
imageSize.height);
```

```
    myLayer = layerContents.CGImage;

    // Add the layer to the view.
    CALayer*    viewLayer = self.view.layer;
    [viewLayer addSublayer:myLayer];

    // Center the layer in the view.
    CGRect        viewBounds = backingView.bounds;
    myLayer.position = CGPointMake(CGRectGetMidX(viewBounds),
CGRectGetMidY(viewBounds));

    // Release the layer, since it is retained by the view's layer
    [myLayer release];
}
```

Defining a Custom View
If the standard system view do not do exactly what you want, you can define a custom view.

> Checklist for Implementing a Custom View
1. The following checklist includes the most important methods you can override when you implement a custom view.
=> Define the appropriate initialization methods for your view:
    > For views you plan to create programmatically, override the initWithFrame: method or define a custom initialization method.
    > For views you plan to load from nib files, override the initWithCoder: method. Use this method to initialize your view and put it into a known state.

=> Implement a dealloc method to handle the cleanup of any custom data.

=> To handle any custom drawing, override the drawRect: method and do your drawing there.

=> Set the autoresizingMask property of the view to define its autoresizing behavior.

=> If your view class manages one or more integral subviews, do the following:
    > Create those subviews during your view’s initialization sequence.

> Set the autoresizingMask property of each subview at creation time.
> If your subviews require custom layout, override the layoutSubviews method and implement your layout code there.

=> To handle touch-based events, do the following:
> Attach any suitable gesture recognizers to the view by using the addGestureRecognizer: method.
> For situations where you want to process the touches yourself, override the touchesBegan:withEvent:, touchesMoved:withEvent:, touchesEnded:withEvent:, and touchesCancelled:withEvent: methods. (Remember that you should always override the touchesCancelled:withEvent: method, regardless of which other touch-related methods you override.)

=> If you want the printed version of your view to look different from the onscreen version, implement the drawRect:forViewPrintFormatter: method.

> Initializing Your Custom View
1. Every new view object you define should include a custom initWithFrame: initializer method.
Initializing a view subclass

```
- (id)initWithFrame:(CGRect)aRect {
    self = [super initWithFrame:aRect];
    if (self) {
        // setup the initial properties of the view
        ...
      }
    return self;
}
```

2. If you plan to load instance of your custom view in a nib file, you should use initWithCoder: method instead.

> Implementing Your Drawing Code
1. You need to override the drawRect: method for you custom drawing.

2. The implementation of your drawRect: method should do one thing only: draw your content.

3. Before calling drawRect: method to draw your content, UIKit configures the basic drawing environment for your view. So you can begin drawing your content using native drawing technologies like UIKit and Core Graphics. You can get a pointer to the current graphics context using the UIGraphicsGetCurrentContext function.
3. A drawing method.

```
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGRect    myFrame = self.bounds;

    // Set the line width to 10 and inset the rectangle by
    // 5 pixels on all sides to compensate for the wider line.
    CGContextSetLineWidth(context, 10);
    CGRectInset(myFrame, 5, 5);

    [[UIColor redColor] set];
    UIRectFrame(myFrame);
}
```

4. You should set the opaque property to YES only if you know the view's content is completely opaque.

5. Another way to improve drawing performance, especially during scrolling, is to set the clearsContextBeforeDrawing property of your view to NO.

> Responding to Events
1. View objects are also responder objects.

2. In addition handling touch events directly, view can also use gesture recognizers to detect common touch-related gestures.

3. If you prefer to handle touch event directly, you can implement the following methods:
=> touchesBegan:withEvent:
=> touchesMoved:withEvent:
=> touchesEnded:withEvent:
=> touchesCancelled:withEvent:

4. If you plan to track multi finger gestures from your view's event-handler methods, you need to enable multitouch events by setting the

multipleTouchEnabled property of your view to YES.

5. Some views, like labels, images, disable event handling. You can control whether a view is able to receive touch event by changing the value of the view's userInteractionEnabled property.

> Cleaning Up After Your View
If your view class allocates any memory, stores references to any custom objects, or holds resources that must be released when the view is released, you must implement a dealloc method.
Implementing the dealloc method

```
- (void)dealloc {
    // Release a retained UIColor object
    [color release];

    // Call the inherited implementation
    [super dealloc];
}
```