

CSIT242

Mobile Application Development

LECTURE 2 - USER INTERFACE DESIGN. VIEWS.
CONTROLLERS.

Outline

1. User Interface Design
2. Android Views and Layouts
3. Event Listeners, Event Handlers
4. iOS Views and Storyboards
5. Outlets, Actions



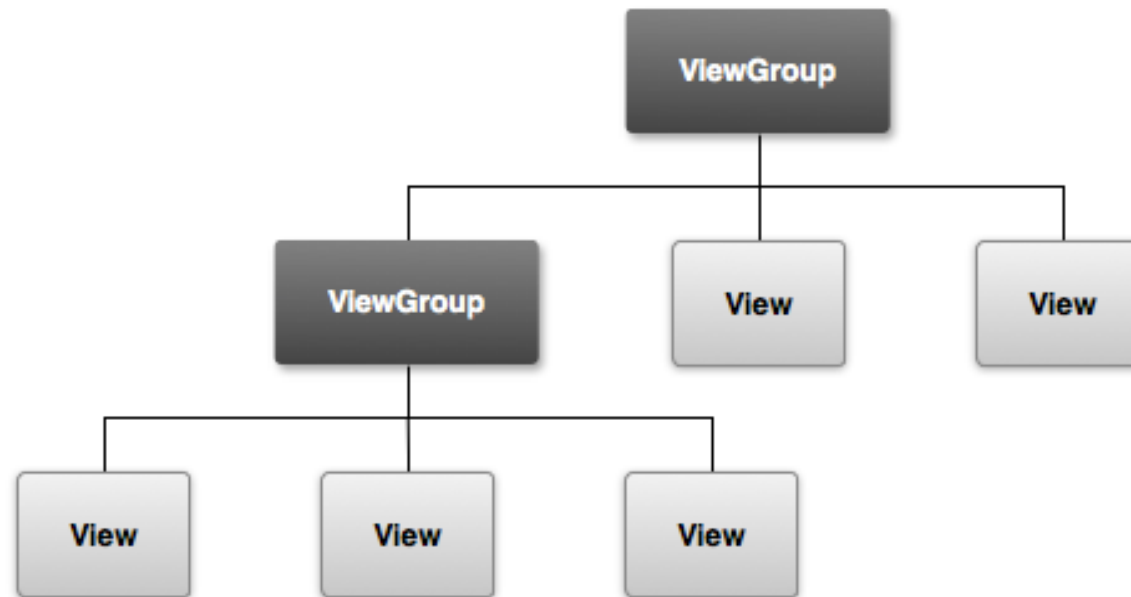
Android – User Interface overview

- App's user interface is everything that the user can see and interact with.
- All user interface elements in an Android app are built using View and ViewGroup (composite view) objects.
 - All UI components are subclasses of Android View class, or ViewGroup.
 - ViewGroups are also called container views.
- A View is an object that draws something on the screen that the user can interact with.
- A ViewGroup is an object (invisible container) that holds other View (and ViewGroup) objects in order to define the position of the views in the interface.



Android – User Interface overview

- The user interface for each component of the app is defined using a hierarchy of View and ViewGroup objects, as shown.
- Each view group is an invisible container that organizes child views (that may be input controls or other widgets that draw some part of the UI).



Source: <https://developer.android.com/guide/topics/ui/declaring-layout>



Android – User Interface overview

- A layout (special type of View called ViewGroup) is container view that defines the structure for a user interface in the app, such as in an activity.
- All elements in the layout are built using a hierarchy of View and ViewGroup objects.
- Android provides a variety of pre-build UI components (or widgets), both View and ViewGroup subclasses, that offer common input controls
 - Button, CheckBox, TextView, ProgressBar, RadioButtons ...
- Android provides a collection of structured layout objects - various layout models (such as constraint, linear or relative layout).
- Android also provides other UI modules for special interfaces such as dialogs, notifications, and menus.



Android – Layouts

- A layout as a visual structure for a user interface can be declared in two ways:
 - XML Layout - Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses. Android studio's Layout Editor can be used for building XML layout using drag-and-drop interface.
 - Instantiate layout elements at runtime - The application can create View and ViewGroup objects (and manipulate their properties) programmatically.
- Using Android's XML vocabulary, allows quick design of UI layouts and the screen elements they contain (in the same way web pages in HTML — with a series of nested elements are created).



Android – User Interface Layout

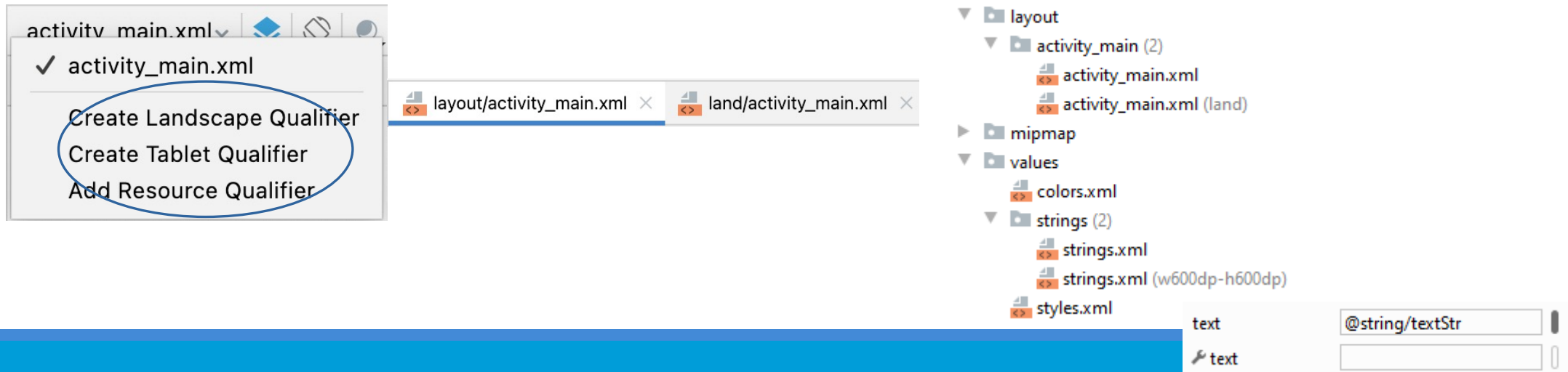
- Each layout file must contain exactly one root element, which must be a View or ViewGroup object.
- Once the root element is defined, additional layout objects or widgets as child elements can be added to gradually build a View hierarchy that defines the layout (tree structure).

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/myTextView"
        android:layout_width="215dp"
        android:layout_height="35dp"
        android:layout_marginTop="60dp"
        android:text="My Text View"
        android:textAlignment="center"
        android:textColor="@color/colorPrimary"
        android:textSize="24sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <Button
        android:id="@+id/myButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="60dp"
        android:text="Click Me"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/myTextView" />
</androidx.constraintlayout.widget.ConstraintLayout>
```



Android – User Interface Layout

- Android studio allows creation of alternative layout resources to optimize the UI design for certain screen sizes:
 - Open your default layout and then click the arrow beside the file tab.
 - In the drop-down list, click to create a suggested qualifier such as *Create Landscape Qualifier* or click *Create Resource Qualifier*.
 - If you selected *Create Resource Qualifier*, in the *Select Resource Directory* select a screen qualifiers and click OK.





Android – Layouts

- When the application is compiled, each XML layout file is compiled into a View resource.
- The layout resource from the application code should be loaded in the *Activity.onCreate()* method implementation.
 - It can be done by calling *setContentView()* method, and passing the reference to the layout resource in the form of:

R.layout.layout_file_name

- For example, if your XML layout is saved as *activity_main.xml*, you would load it for your Activity like:

```
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```



Android – Layout Attributes

- Every View and ViewGroup object supports their own variety of XML attributes.
- Some attributes are specific to a View object (for example, TextView supports the textSize attribute), but these attributes are also inherited by any View objects that may extend this class.

- ID - Any View object may have an integer ID associated with it, to uniquely identify the View within the tree.

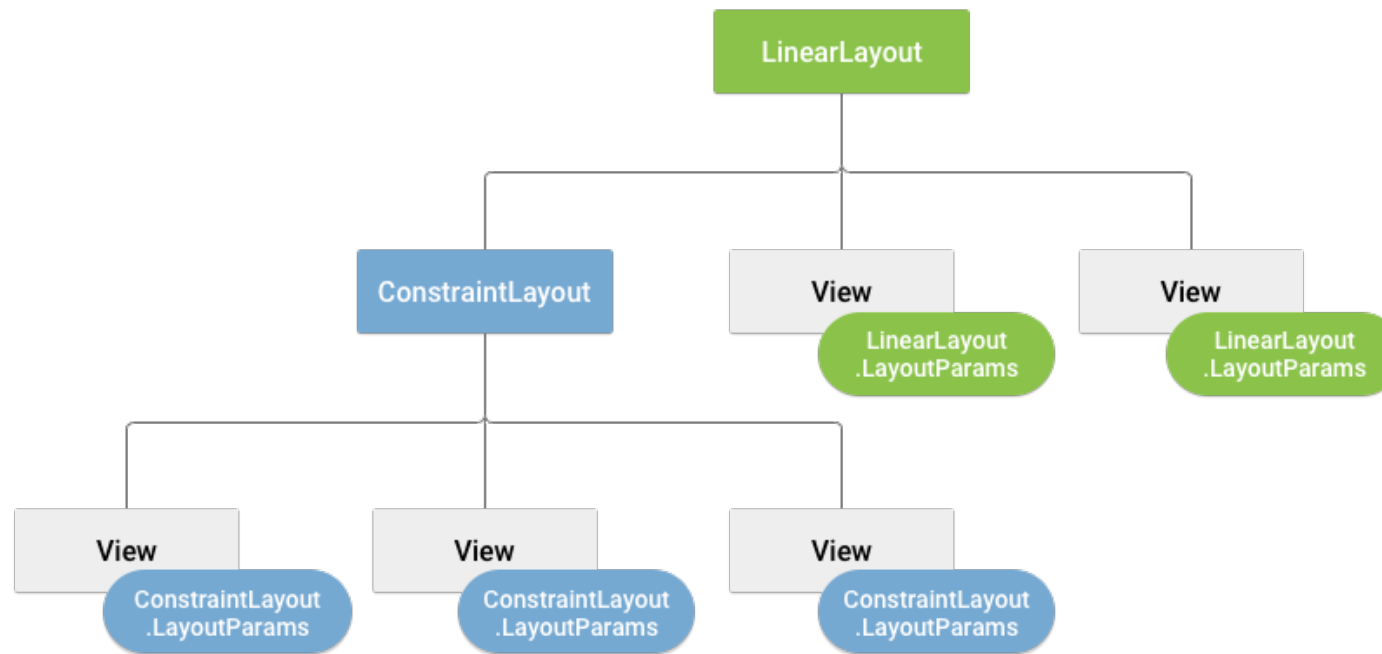
`android:id="@+id/my_button"`

- Layout Parameters - XML layout attributes *named layout_something* define layout parameters for the View that are appropriate for the ViewGroup in which it resides.



Android – Layout Parameters

- Every *ViewGroup* class implements a nested class that extends *ViewGroup.LayoutParams*.
- This subclass contains property types that define the size and position for each child view, as appropriate for the view group.



Source: <https://developer.android.com/guide/topics/ui/declaring-layout>



Android – Layout Parameters

- All view groups include a width and height (*layout_width* and *layout_height*), and each view is required to define them.
- Many LayoutParams also include optional margins and borders.
- Width and height can be specified with exact measurements or more often, can be used one of these constants:
 - *wrap_content* tells the view to size itself to the dimensions required by its content.
 - *match_parent* (named *fill_parent* before API Level 8) tells the view to become as big as its parent view group will allow.



Android – Layout Position

- The geometry of a view is that of a rectangle.
- A view has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height.
- The unit for location and dimensions is the pixel.
- It is possible to retrieve the location of a view by invoking the methods *getLeft()* and *getTop()*.
 - *getLeft()* returns the left, or X, coordinate of the rectangle representing the view.
 - *getTop()* returns the top, or Y, coordinate of the rectangle representing the view.



Android – Common Layouts

- Each subclass of the ViewGroup class provides a unique way to display the views nested within it.
- Common layout types that are built into the Android platform are:

- Linear Layout,
- Relative Layout,
- Web View.

Linear Layout



Relative Layout



Web View



Source: <https://developer.android.com/guide/topics/ui/declaring-layout>

- In Android 7 and later, the recommended layout is **Constraint Layout**.

Although you can nest one or more layouts within another layout to achieve your UI design, you should strive to keep your layout hierarchy as shallow as possible. Your layout draws faster if it has fewer nested layouts (a wide view hierarchy is better than a deep view hierarchy).



Android – Linear Layout

- LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.
- The layout direction can be specified with the `android:orientation` attribute.
- All children of a LinearLayout are stacked one after the other, so:
 - a vertical list will only have one child per row, no matter how wide they are, and
 - a horizontal list will only be one row high (the height of the tallest child, plus padding).
- A LinearLayout supports assigning a weight to individual children (*layout_weight* attribute).
 - This attribute assigns an “importance” value to view in terms of how much space it should occupy on the screen.
- A LinearLayout respects margins between children and the gravity (right, center, or left alignment) of each child.



Android – Relative Layout

- RelativeLayout is a view group that displays child views in relative positions.
- The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent RelativeLayout area (such as aligned to the bottom, left or center).
 - The two elements can be aligned by right border, or make one below another, centered in the screen, centered left, and so on.



Android – Constraint Layout

- ConstraintLayout is a view group introduced in Android 7 (Android Studio 2.3).
- ConstraintLayout allows the positioning and behaviour of the views in a layout to be defined by simple constraint settings assigned to each child view.
 - It gives instructions for how each view (in relation to Layout and other views) should be displayed.
- The flexibility of this layout allows complex layouts to be quickly and easily created without the necessity to nest other layout types inside each other - resulting in improved layout performance.
- ConstraintLayout is tightly integrated into the Android Studio Layout Editor tool (design editor's blueprint tool).



Android – Dynamic Layouts

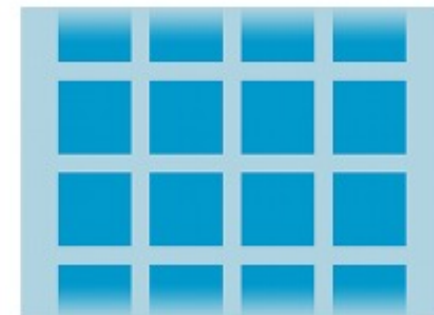
- When the content for the layout is dynamic or not pre-determined, can be used a layout that subclasses AdapterView to populate the layout with views at runtime.
- A subclass of the AdapterView class uses an Adapter to bind data to its layout.
- The Adapter behaves as a middleman between the data source and the AdapterView layout—the Adapter retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the AdapterView layout.
- Common layouts backed by an adapter include:

List View



Displays a scrolling single column list.

Grid View



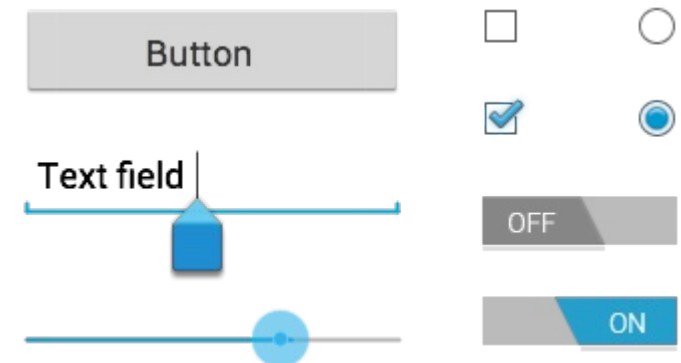
Displays a scrolling grid of columns and rows.

Source: <https://developer.android.com/guide/topics/ui/declaring-layout>



Android – Input Controls

- Input controls are the interactive components in the app's user interface.
 - buttons, text fields, seek bars, checkboxes, zoom buttons, toggle buttons, and many more.
- Adding an input control to the UI is as simple as adding an XML element to your XML layout.
- Or the control can be drag-and-drop on the design window.



Source: https://www.tutorialspoint.com/android/android_user_interface_controls.htm



Android – Common Controls

Control Type	Description
Button	A push-button that can be pressed, or clicked, by the user to perform an action.
Text field	An editable text field or present text to the user.
Checkbox	An on/off switch that can be toggled by the user. Usually used for a group of selectable options that are not mutually exclusive.
Radio button	Similar to checkboxes, but used for cases where only one option can be selected in the group.
Toggle button	An on/off button - changing a setting between two states.
Spinner	A drop-down list that allows users to select one value from a set.
Pickers	For example: DatePicker or TimePicker.



Android – Custom Components

- Android offers a sophisticated and powerful componentized model for building your UI, based on the fundamental layout classes: View and ViewGroup, but also allows creation of custom components.
- Basic approach to Custom components - extend an existing View class, then override some of the methods from the superclass and use your new extension class.
- Fully customized components can be used to create graphical components that appear however you wish. Perhaps a graphical VU meter that looks like an old analog gauge, or a sing-a-long text view where a bouncing ball moves along the words so you can sing along with a karaoke machine.



Android – Input Events

- Events in Android can take a variety of different forms, but are usually generated in response to an external action.
- The most common form of events involve some form of interaction with the touch screen. Such events fall into the category of input events.
- The Android framework maintains an event queue into which events are placed as they occur - first-in, first-out (FIFO) basis.
- In order to be able to capture the user interaction with the UI the view must have in place an event listener.
- In order to handle the event that it has been passed, the view must have in place an callback method.



Android – Event Listeners and CallBack Methods

- The Android View class, from which all user interface components are derived, contains a range of event listener interfaces, each of which contains an abstract declaration for a callback method.
- In order to be able to respond to an event of a particular type, a view must register the appropriate event listener and implement the corresponding callback method.
 - For instance, when a View (such as a Button) is touched, the onTouchEvent() method is called on that object.



android:onClick Resource

- In Android is available a shortcut for calling a callback method for capturing click events (user “clicks” on a button view in the user interface).
- For example, if there is a button view named *button1* with the requirement that when the user touches the button, a method called *buttonClick()* declared in the activity class is called, can be done by adding a single line to the declaration of the button view in the XML file.

```
<Button
    android:id="@+id/myButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="60dp"
    android:onClick="clickButton"
    android:text="Click Me"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/myTextView" />
```

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void clickButton(View v) {
        TextView statusText = (TextView) findViewById(R.id.myTextView);
        statusText.setText("Button clicked");
    }
}
```




Android – Event Listeners and CallBack Methods

- An event listener is an interface in the View class that contains a single callback method.
- These methods are called by the Android framework when the View to which the listener has been registered is triggered by user interaction with the item in the UI.
- Common event listeners with associated callback methods:
 - **onClickListener** – Used to detect click style events whereby the user touches and then releases an area of the device display occupied by a view. Corresponds to the **onClick()** callback method which is passed a reference to the view that received the event as an argument.
 - **onLongClickListener** – Used to detect when the user maintains the touch over a view for an extended period. Corresponds to the **onLongClick()** callback method which is passed as an argument the view that received the event.



Android – Event Listeners and CallBack Methods

- **onTouchListener** – Used to detect any form of contact with the touch screen including individual or multiple touches and gesture motions. Corresponding with **the onTouch()** callback. The callback method is passed as arguments the view that received the event and a MotionEvent object.
- **onCreateContextMenuListener** – Listens for the creation of a context menu as the result of a long click. Corresponds to the **onCreateContextMenu()** callback method. The callback is passed the menu, the view that received the event and a menu context object.
- **onFocusChangeListener** – Detects when focus moves away from the current view as the result of interaction with a track-ball or navigation key. Corresponds to the **onFocusChange()** callback method which is passed the view that received the event and a Boolean value to indicate whether focus was gained or lost.
- **onKeyListener** – Used to detect when a key on a device is pressed while a view has focus. Corresponds to the **onKey()** callback method. Passed as arguments are the view that received the event, the KeyCode of the physical key that was pressed and a KeyEvent object.



Android – Event Listeners and CallBack Methods

```
import androidx.appcompat.app.AppCompatActivity;
import android.widget.Button;
import android.widget.TextView;
import android.view.View;
import android.os.Bundle;
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = (Button) findViewById(R.id.myButton);
        button.setOnClickListener(
            new Button.OnClickListener() {
                public void onClick(View v) {
                    TextView statusText = (TextView) findViewById(R.id.myTextView);
                    statusText.setText("Button clicked");
                }
            });
        button.setOnLongClickListener(
            new Button.OnLongClickListener() {
                public boolean onLongClick(View v) {
                    TextView statusText = (TextView) findViewById(R.id.myTextView);
                    statusText.setText("Long button click");
                    return true; //return false; }
            });
    }
}
```



Android – Event Listeners and CallBack Methods

- Regarding the previous example
 - When a long click is detected, the `onLongClick()` callback method will display “Long button click” on the text view. But the callback method also returns a value of `true` to indicate that it has consumed the event. In this case the `onClick` listener code will not be called.
 - If the return value is `false`, it will be indicated to the Android framework that the event was not consumed by the method and was eligible to be passed on to the next registered listener on the view. In that case the `onClick` listener code (as next registered listener on the view) will be called.



Android – Event Handlers

- When we are building a custom component from View, we'll be able to define several callback methods used as default event handlers.
 - `onKeyDown(int, KeyEvent)` - Called when a new key event occurs.
 - `onKeyUp(int, KeyEvent)` - Called when a key up event occurs.
 - `onTrackballEvent(MotionEvent)` - Called when a trackball motion event occurs.
 - `onTouchEvent(MotionEvent)` - Called when a touch screen motion event occurs.
 - `onFocusChanged(boolean, int, Rect)` - Called when the view gains or loses focus.
- Also, there are some other methods which are not part of the View class and can directly impact the way of handling events
 - `Activity.dispatchTouchEvent(MotionEvent)` - This allows Activity to intercept all touch events before they are dispatched to the window.
 - `ViewGroup.onInterceptTouchEvent(MotionEvent)` - This allows a ViewGroup to watch events as they are dispatched to child Views.
 - `ViewParent.requestDisallowInterceptTouchEvent(boolean)` - This allows a parent View to indicate that it should not intercept touch events with `onInterceptTouchEvent(MotionEvent)`.



Android – Handling State Change

- Any configuration change (such as rotating the orientation of the device's screen) that impacts the appearance of an activity will cause the activity to be destroyed and recreated.
- The app can react to those changes by taking appropriate actions.
- Activity state:
 - Persistent state - data that needs to be saved to a database, content provider or file,
 - Dynamic state - the appearance of the user interface (such as text entered into a text field but not yet committed to the app's data model).
- The Activity and Fragment classes contain a methods that can handle the state change and preserve the dynamic state:
 - `onConfigurationChanged()` – Called when a configuration change occurs for which the activity has indicated it is not to be restarted.
 - `onRestoreInstanceState(Bundle savedInstanceState)` – This method is called immediately after a call to the `onStart()` method in the event that the activity is restarting from a previous invocation in which state was saved.
 - `onSaveInstanceState(Bundle outState)` – Called before an activity is destroyed so that the current dynamic state (usually relating to the user interface) can be saved.

iOS – Application (review)

- An iOS application is created by combining multiple elements:
 - Essential Files (Icon, default & info.plist)
 - Interface Builder documents (.xib, .storyboard)
 - Code files (.h, .m, .swift)
 - Frameworks, Libraries & Workspaces
 - Supporting Files (Sounds, Images, Video, Pictures, databases)
 - Application Binary / Executable (e.g. HelloWorld.app)
- Everything required by the application is in the **bundle**.

iOS – Views

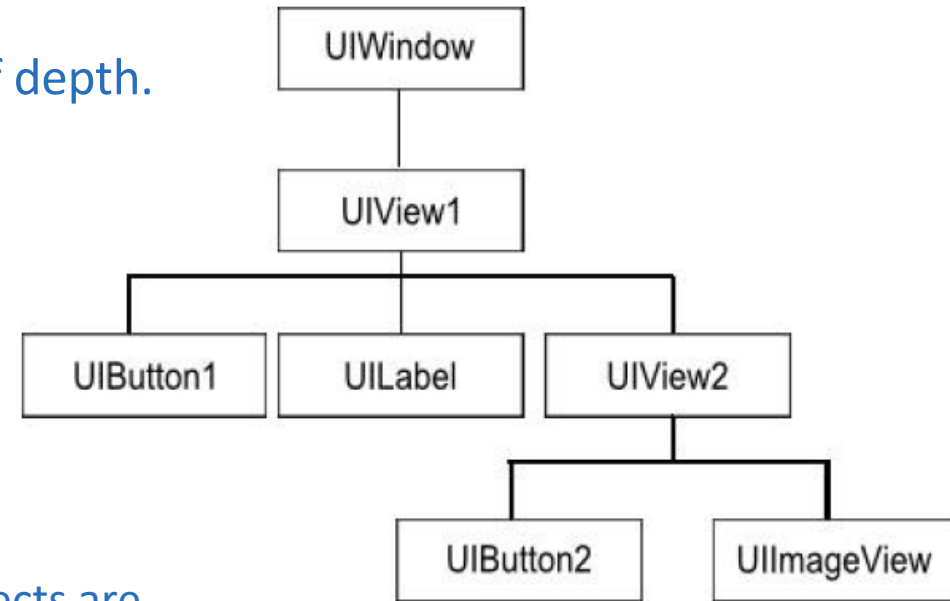
- Views are visual objects that are assembled to create the user interface of an iOS application.
- They essentially define what happens within a specified rectangular area of the screen, both visually and in terms of user interaction.
- All views are subclasses of the UIView class and include items (such as label – UILabel, image view - UIImageView) and controls (such as the button –UIButton, text field - UITextField).
- A view is also a responder (UIView is a subclass of UIResponder) - is subject to user interactions, such as taps and swipes.

iOS – Views

- In iOS, the UIWindow class provides the surface on which the view components are displayed.
- iOS app, for iPhones, typically only has one window that fill the entire screen and it lacks the title bar.
- UIWindow is also a subclass of the UIView class and sits at the root of the view hierarchy.
 - The user does not see or interact directly with the UIWindow object.
 - It may be created programmatically, or automatically by Interface Builder when the user interface is designed.

iOS – Views

- iOS user interfaces are constructed using a hierarchical approach whereby different views are related through a parent/child relationship.
- At the top of this hierarchy sits the UIWindow object.
 - Other views can be added to the hierarchy.
 - View hierarchies can be nested to any level of depth.
 - A subview can only have one direct parent.
- For example:
 - UIWindow object is the parent or superview of the UIView1 instance, and
 - UIView1 is the child, or subview of the UIWindow.
 - Similarly, the UIView2, UILabel and UIButton1 objects are all subviews of the UIView1.



iOS – View Types



















































View Type	Description
The Window	The UIWindow is the root view of the view hierarchy and provides the surface on which all subviews draw their content.
Container Views	Enhance the functionality of other view objects. For example, the UIScrollView class provides scrollbars and scrolling functionality for the UITableView and UITextView classes; the UIToolbar view serves to group together multiple controls in a single view.
Controls	The controls category encompasses views that both present information and respond to user interaction. Control views inherit from the UIControl class (itself a subclass of UIView) and include items such as buttons, sliders and text fields.

iOS – View Types

View Type	Description
Display Views	Display views are similar to controls in that they provide visual feedback to the user, the difference being that they do not respond to user interaction (such as UILabel and UIImageView classes).
Text and Web Views	The UITextView and UIWebView classes both fall into this category and are designed to provide a mechanism for displaying formatted text to the user.
Navigation Views and Tab Bars	Navigation views and tab bars provide mechanisms for navigating through an application user interface. They work in conjunction with the view controller and are typically created from within Interface Builder.
Alert Views	Views in this category are designed specifically for prompting the user with urgent or important information together with optional buttons to call the user to action.

UIWebView has been deprecated by Apple since iOS 8 and is no longer recommended for use in new apps. In apps that run in iOS 8 and later, use the WKWebView class instead of using UIWebView.

iOS – Basic Interface Elements

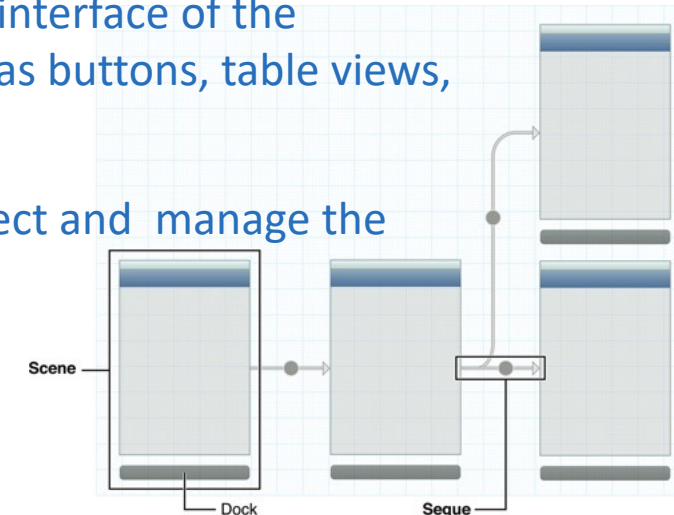
Label Label – A variably sized amount of static text.	 Table View Cell – Defines the attributes and behavior of cells (rows) in a table view.	 MapKit View – Displays maps and provides an embeddable interface to navigate map content.	 Long Press Gesture Recognizer – Provides a recognizer for long press gestures which are invoked on the view.	 AVKit Player View Controller – A view controller that manages a AVPlayer object.
Button Button – Intercepts touch events and sends an action message to a target object when it's tapped.	 Image View – Displays a single image, or an animation described by an array of images.	 GLKit View – Provides a default implementation of an OpenGL ES-aware view.	 View Controller – A controller that supports the fundamental view-management model in iOS.	 Navigation Bar – Provides a mechanism for displaying a navigation bar just below the status bar.
 Segmented Control – Displays multiple segments, each of which functions as a discrete button.	 Collection View – Displays data in a collection of cells.	 iAd BannerView – The ADBannerView class provides a view that displays banner advertisements to the user.	 Navigation Controller – A controller that manages navigation through a hierarchy of views.	 Navigation Item – Represents a state of the navigation bar, including a title.
 Text Field – Displays editable text and sends an action message to a target object when Return is tapped.	 Collection View Cell – Defines the attributes and behavior of cells in a collection view.	 SceneKit View – A view for displaying a 3D scene.	 Table View Controller – A controller that manages a table view.	 Toolbar – Provides a mechanism for displaying a toolbar at the bottom of the screen.
 Slider – Displays a continuous range of values and allows the selection of a single value.	 Collection Reusable View – Defines the attributes and behavior of reusable views in a collection view, such as a section header or footer.	 Web View – Displays embedded web content and enables content navigation.	 Tab Bar Controller – A controller that manages a set of view controllers that represent tab bar items.	 Bar Button Item – Represents an item on a UIToolbar or UINavigationController object.
 Switch – Displays an element showing the boolean state of a value. Allows tapping the control to toggle the value.	 Text View – Displays multiple lines of editable text and sends an action message to a target object when Return is tapped.	 Tap Gesture Recognizer – Provides a recognizer for tap gestures which land on the view.	 Split View Controller – A composite view controller that manages left and right view controllers.	 Tab Bar – Provides a mechanism for displaying a tabs at the bottom of the screen.
 Activity Indicator View – Provides feedback on the progress of a task or process of unknown duration.	 Scroll View – Provides a mechanism to display content that is larger than the size of the application's window.	 Pinch Gesture Recognizer – Provides a recognizer for pinch gestures which are invoked on the view.	 Page View Controller – Presents a sequence of view controllers as pages.	 Tab Bar Item – Represents an item on a UITabBar object.
 Progress View – Depicts the progress of a task over time.	 Date Picker – Displays multiple rotating wheels to allow users to select dates and times.	 Rotation Gesture Recognizer – Provides a recognizer for rotation gestures which are invoked on the view.	 GLKit View Controller – A controller that manages a GLKit view.	 Search Bar – Displays an editable search bar, containing the search icon, that sends an action message to a target object when Return is tapped.
 Page Control – Displays a dot for each open page in an application and supports sequential navigation through the pages.	 Picker View – Displays a spinning-wheel or slot-machine motif of values.	 Swipe Gesture Recognizer – Provides a recognizer for swipe gestures which are invoked on the view.	 Object – Provides a template for objects and controllers not directly available in Interface Builder.	
 Stepper – Provides a user interface for incrementing or decrementing a value.	 Visual Effect View with Blur – Provides a blur effect	 Pan Gesture Recognizer – Provides a recognizer for panning (dragging) gestures which are invoked on the view.	 External Object – Provides a placeholder for an object that exists outside of the document.	
 Table View – Displays data in a list of plain, sectioned, or grouped rows.	 Visual Effect Views with Blur and Vibrancy – Provides a blur effect, plus vibrancy for nested views	 Screen Edge Pan Gesture Recognizer – Provides a recognizer for panning (dragging) gestures which are invoked on the view and start near an edge of the screen.	 Collection View Controller – A controller that manages a collection view.	

iOS – Storyboards & .nib files

- There are three types of user interface design approaches:
 - iOS Storyboards,
 - NIBs (or XIBs),
 - Custom Code (no GUI tools, but rather, handling all custom positioning, animation, etc. programmatically).

iOS – Storyboards

- iOS Storyboards are visual tool for laying out multiple application interfaces (screens) and the transitions between them.
 - A storyboard is a visual representation of the user interface of an iOS application, showing screens of content and the connections between those screens.
 - A storyboard is composed of a sequence of scenes, each of which represents a view controller and its views; scenes are connected by segue objects, which represent a transition between two view controllers.
 - Xcode provides a visual editor for storyboards, where the user interface of the application can be lay out and designed by adding views (such as buttons, table views, and text views) onto scenes.
 - A storyboard enables connection of a view to its controller object and manage the transfer of data between view controllers.
 - Using storyboards enable visualization of the appearance and flow of the user interface on one canvas.



Source: https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html#//apple_ref/doc/uid/TP40009071-CH99-SW1

iOS – .nib (.xib) files

- A nib or xib file is a special type of resource file that can be used to store the user interfaces of iOS.
 - Each NIB file corresponds to a view element and can be laid out in the Interface.
 - Interface Builder can be used to design the visual parts of the app - such as windows and views - and sometimes to configure nonvisual objects, such as the controller objects that the app uses to manage its windows and views.
 - As the Interface Builder document is edited, an object graph is created and archived when the file is saved.

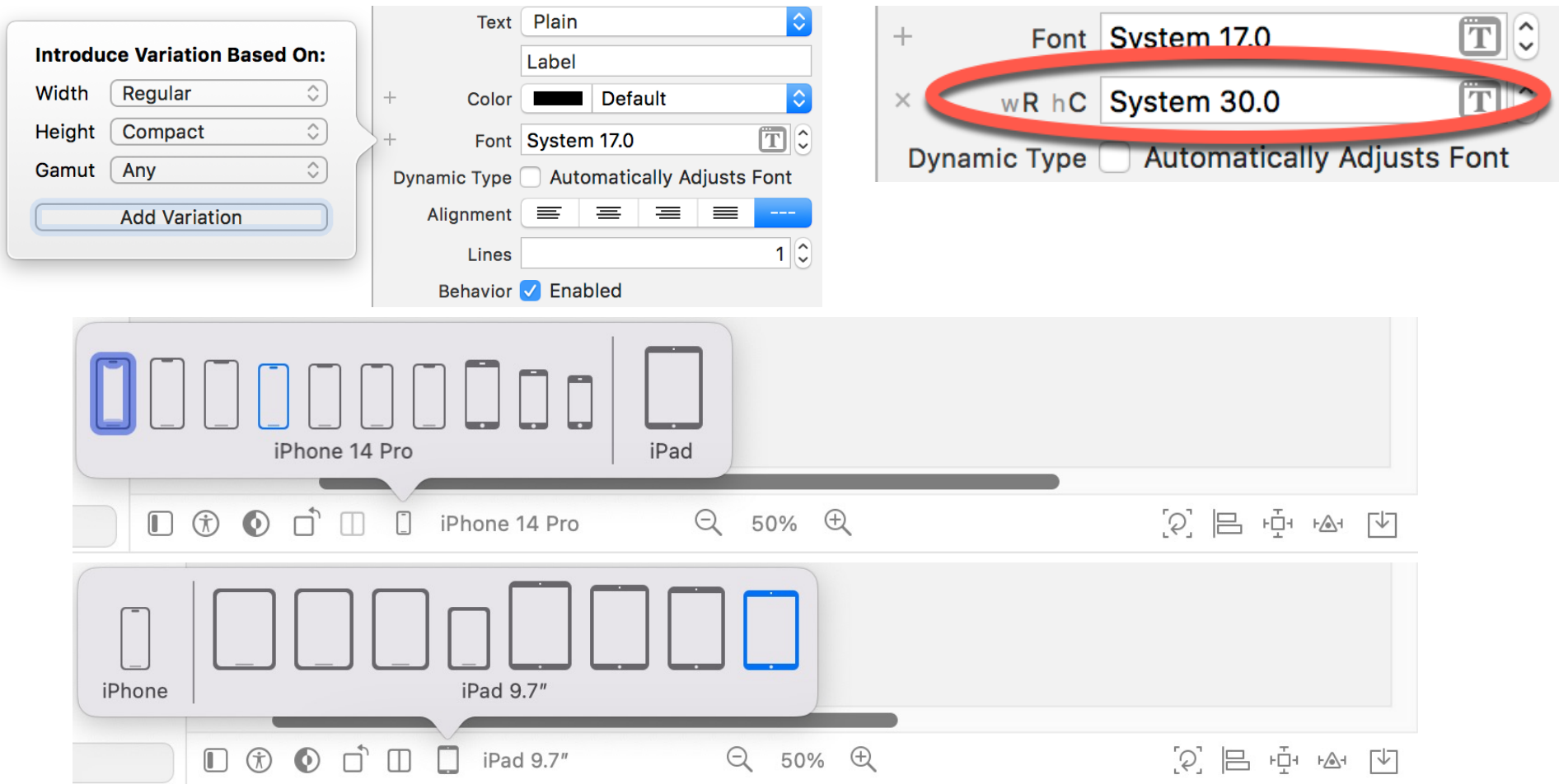
iOS – Auto Layout

- Layout must be able to adapt to variables such as device rotating, dynamic changes to content and differences in screen resolution and size.
- Both iOS and OS X include a powerful layout system called Auto Layout, with excellent support built into Interface Builder.
- Auto Layout is based on the idea that each object in the interface can define a constraint to control how it reacts to the parent view and other interface controls.
 - For example, you can prioritize whether a button stays a specific size or expands to accommodate larger text when displaying a different language.
- You can also take direct control of constraints to define the exact priority of each, defining how the app will work on different screen sizes, when rotated, or when run in new locales.

iOS – Traits and Size Classes

- iOS 9 and Xcode 7 introduced the concepts of traits and size classes.
- Traits and size classes allow a user interface layout to be designed for multiple screen sizes and orientations within a single storyboard file.
- Traits define the features of the environment an app is likely to encounter when running on an iOS device.
- Most powerful trait category relates to the size and orientation of the device screen. These trait values are referred as size classes.
- Size classes categorize the various screen areas that an application user interface is likely to encounter during execution.
 - Size classes represent width (w) and height (h) of the device in the term of being compact (C) - denotes constrained space, or regular (R) - denotes expansive space.
 - For example: iPhones in portrait are represent with wC hR, and in landscape with wC hC. Plus, XR or XS Max version in landscape are represented by wR hC

iOS – Traits and Size Classes



iOS – Table Views

- Table Views are the cornerstone of the navigation system for many iOS iPhone applications.
- Table Views present the user with data in a list format and are represented by the UITableView class of the UIKit framework.
 - The data is presented in rows, whereby the content of each row is implemented in the form of a UITableViewCell object.
 - Each table cell can display a text label (textLabel), a subtitle (detailedTextLabel) and an image (imageView).
 - More complex cells can be created by either adding subviews to the cell, or subclassing UITableViewCell and adding custom functionality and appearance.

iOS – Table Views

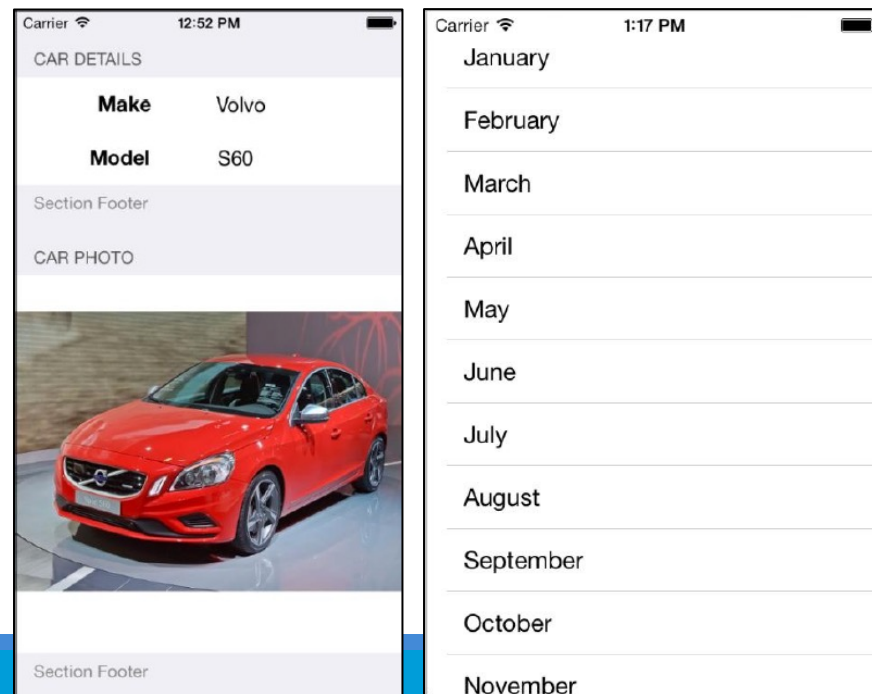
- Static tables are useful in situations when a fixed number of rows need to be displayed in a table.
 - The settings page for an application, for example, would typically have a predetermined number of configuration options and would be an ideal candidate for a static table.
- Dynamic tables (also known as prototype-based tables) are intended for use when a variable number of rows need to be displayed from a data source.
 - Within the storyboard editor, Xcode allows you to visually design a prototype table cell which will then be replicated in the dynamic table view at runtime in order to display data to the user.

iOS – Table Views

- Each table view in an application needs to have a delegate and a dataSource associated with it (with the exception of static tables which do not have a data source).
 - The dataSource implements the UITableViewDataSource protocol, which basically consists of a number of methods that define title information, how many rows of data are to be displayed, how the data is divided into different sections and, most importantly, supplies the table view with the cell objects to be displayed.
 - The delegate implements the UITableViewDelegate protocol and provides additional control over the appearance and functionality of the table view including detecting when a user touches a specific row, defining custom row heights and indentations and also implementation of row deletion and editing functions.

iOS – Table Views

- Table views may be configured to use either plain or grouped style.
 - In the grouped style, the rows are grouped together in sections separated by optional headers and footers.
 - In the case of the plain style, the items are listed without separation and using the full width of the display.



Source: iOS 12 App Development Essentials, Neil Smyth, Payload Media, Inc., 2018

iOS – Outlets

- Interface object can be connected with the code (in the source code files) by declaring an instance of it as an IBOutlet.
- @IBOutlet allows user interface object from IB to be “connect” to the code.
- For example

```
@IBOutlet weak var billAmountLabel: UILabel!
```

now we can call the label, and set the text remotely from the code

```
billAmountLabel.text = "$1300.00"
```


iOS – Actions

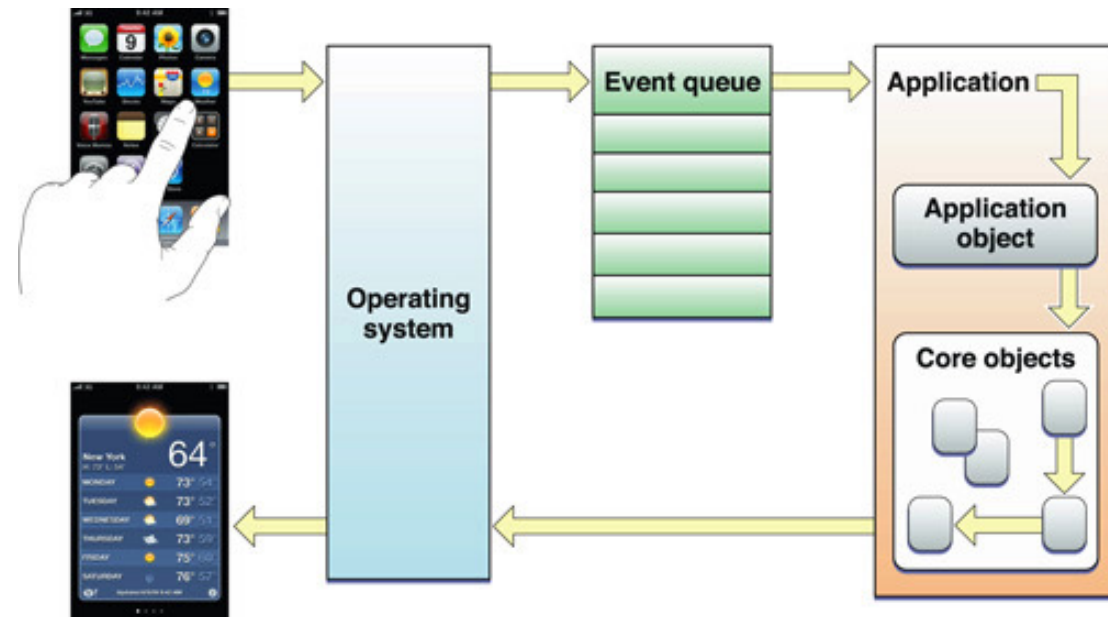
- Actions are methods that are tagged with @IBAction, which tells Interface Builder that this method can be triggered by a UI control in a storyboard or nib file.
- The declaration for an action method (in the source code files) is:

```
@IBAction func doSomething()  
@IBAction func doSomething(sender: UIButton)  
@IBAction func doSomething(sender: UIButton, forEvent event: UIEvent)
```

- The actual name of the method can be anything and it must either take no arguments, take a single argument, usually called sender, or take two arguments: sender and event.
 - When the action method is called, sender will contain a pointer to the object that called it.
 - For example, if this action method was triggered when the user tapped a button, sender would point to the button that was tapped. The sender argument exists so that you can respond to multiple controls using a single action method giving you a way to identify which control called the action method.
- An action method can be declared with a sender argument and then ignore it.

iOS – App's Main Run Loop

- An app's main run loop processes all user-related events.
- The UIApplication object sets up the main run loop at launch time and uses it to process events and handle updates to view-based interfaces.
- As the name suggests, the main run loop executes on the app's main thread.
- This behavior ensures that user-related events are processed serially in the order in which they were received.



Source:

https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedi-a-CocoaApp/MainEventLoop.html#//apple_ref/doc/uid/TP40009071-CH18-SW1

iOS – Event Types

- Apps receive and handle events using responder objects.
 - When the app receives an event, UIKit automatically directs that event to the most appropriate responder object - first responder.
- A responder object is any instance of the UIResponder class, and common subclasses like UIView, UIViewController, and UIApplication.
 - Responders receive the raw event data and must either handle the event or forward it to another responder object.

Event type	First responder
Touch events	The view in which the touch occurred.
Press events	The object that has focus.
Shake-motion events	The object that app designer (or UIKit) designate.
Remote-control events	The object that that app designer (or UIKit) designate.
Editing menu messages	The object that that app designer (or UIKit) designate.
Motion events related to the accelerometers, gyroscopes, and magnetometer	Do not follow the responder chain. Core Motion delivers those events directly to the designated object.

Resources

- Android Studio 4.1 Essentials - Java Edition, Neil Smyth, Payload Media Inc., 2020.
- iOS 17 App Development Essentials, Neil Smyth, Payload Media, Inc., 2023.
- Programming iOS 14: Dive Deep into Views, View Controllers, and Frameworks, Matt Neuburg, O'Reilly Media Inc., 2020.
- <https://developer.android.com/guide/topics/ui/declaring-layout.html>
- <https://developer.android.com/guide/topics/ui/ui-events.html>
- <https://developer.apple.com/library/content/referencelibrary/GettingStarted/DevelopiOSAppsSwift/index.html>
- https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UsingInterfaceBuilder.html