Ubiquitous Computing

COMP H4025

Lecturer: Simon McLoughlin

Lecture 2

Aclknowledgements: Notes compiled from various sources including, android developer website, M. Stepp (CS Stanford), books and my own head.

Lecture Overview

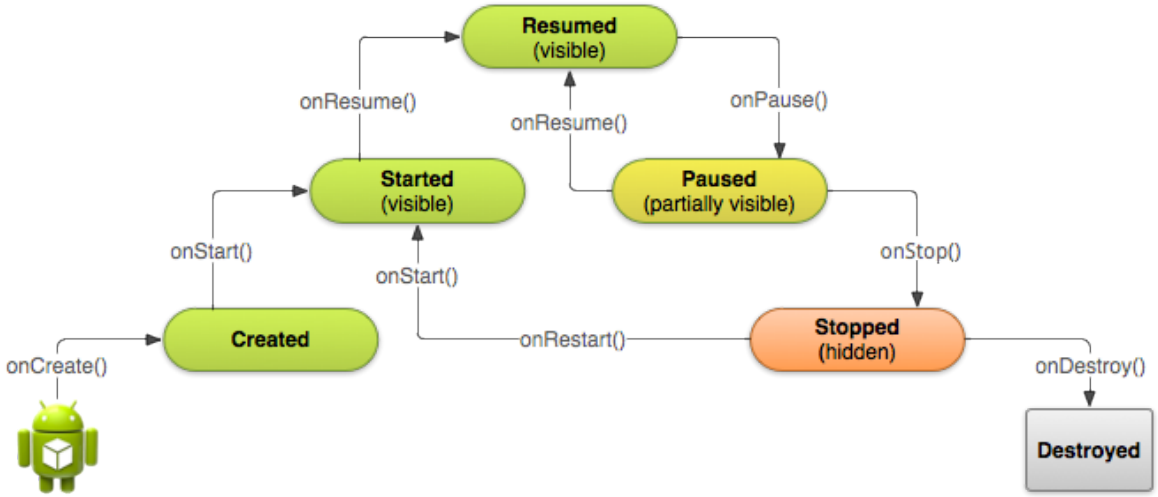
This week:

- More on activities
- User Interfaces
- Layouts
- Menus
- Intents
- Events
- Putting it all together

Choose a paper that sparks your interest on HCI for Ubiquitous Computing/Mobile Computing (use IEEExplore). Read it and write a report on it (5%).

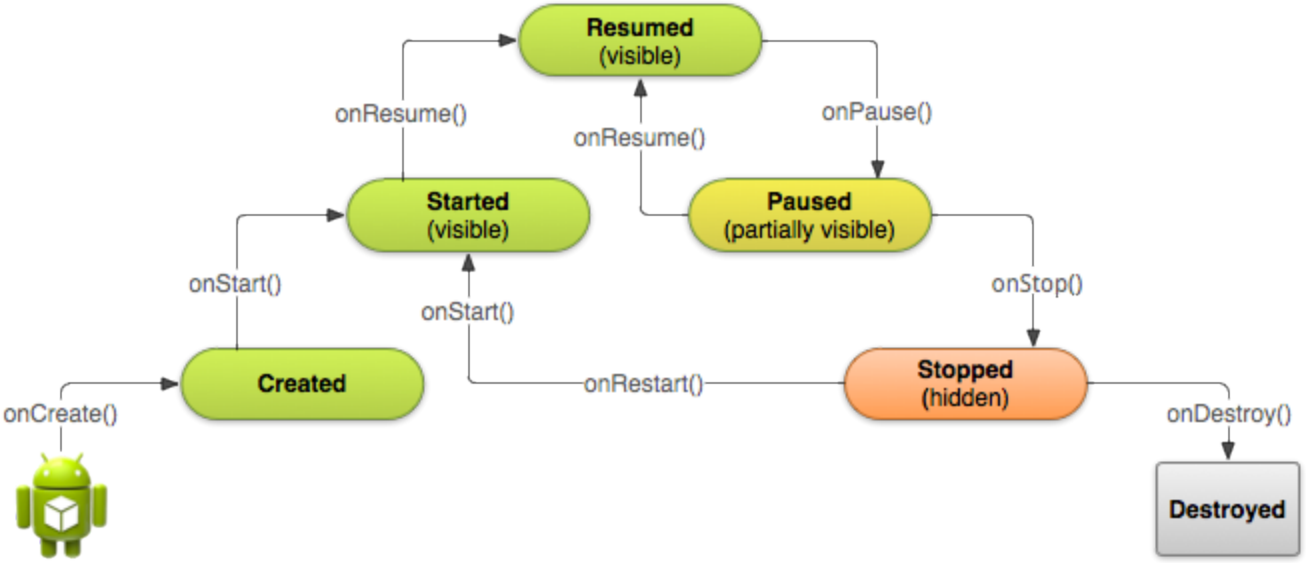
The Activity Life Cycle

- Let us return now to the onCreate method of the Activity and what it means in terms of the Activity life cycle.
- Activities can be in a number of states as depicted below. There are a number of callbacks that cause the state transitions. There are certain things that should happen in these callbacks
- OnCreate callback: perform basic application startup logic that should happen only once for the entire life of the activity. For example, your implementation of onCreate() should define the user interface and possibly instantiate some class-scope variables.
- You must override the onCreate callback. Even if you do not override the onStart() and onResume() callbacks the system will call them anyway



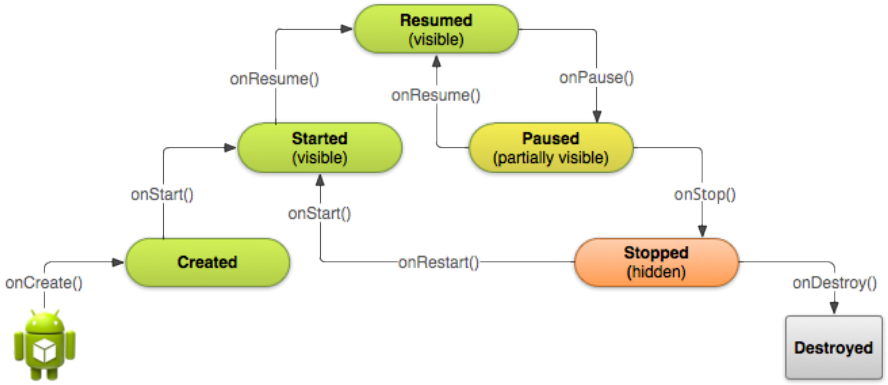
The Activity Life Cycle

- The activity will become paused when a new activity comes to the fore and the old activity is still partially visible in the background (i.e. it loses focus)
- This allows you to stop ongoing actions that should not continue while paused (such as a video/animations) or persist any information that should be permanently saved in case the user continues to leave your app (e.g. draft emails). If the user returns to your activity from the paused state, the system resumes it and calls the onResume() method. At this point you should re-initialise anything suspended in onPause().



The Activity Life Cycle

- onStop() and onRestart() allow you to specifically handle how your activity handles being stopped and restarted. Unlike the paused state, which identifies a partial UI obstruction, the stopped state guarantees that the UI is no longer visible and the user's focus is in a separate activity (or an entirely separate app).
- When your activity receives a call to the onStop() method, it's no longer visible and should release almost all resources that aren't needed while the user is not using it. Once your activity is stopped, the system might destroy the instance if it needs to recover system memory.
- It's uncommon that an app needs to use onRestart() to restore the activity's state, so there aren't any guidelines for this method that apply to the general population of apps, onStart() is called anyway.



Activity States

- The onCreate method should be used to inflate the user interface, allocate class variables, create services and Threads.
- The onCreate method is passed a Bundle object containing the UI state saved the last time the Activity was active. This Bundle is used to restore the user interface to its previous state within the onCreate method by calling onCreate in the super class.
- Override onDestroy to clean up any resources created in onCreate, and ensure that all external connections, such as network or database links, are closed. This is the last chance to clean up resources that might lead to a memory leak.

User Interfaces – Declarative vs Programmatic

- We have seen already how to create a user interface in Android using XML, this is the declarative approach.
- With this approach you can use WYSIWYG tools that generate the XML for you
- You could if you wished adopt a purely programmatic approach like in Java Swing but this is a lot of effort.
- The best practice is to use both. Use the declarative (XML) approach to declare everything about the user interface that is static, such as the layout of the screen, all the widgets, etc. Then switch to a programmatic (Java) approach to define what goes on when the user interacts with the various widgets in the user interface.

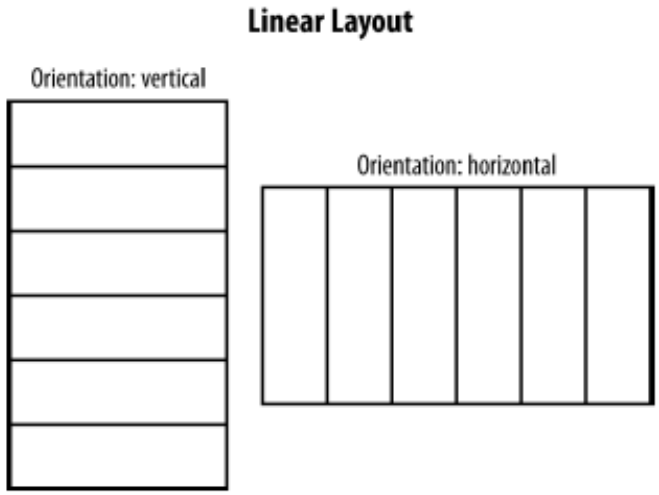
User Interfaces – Views and Layouts

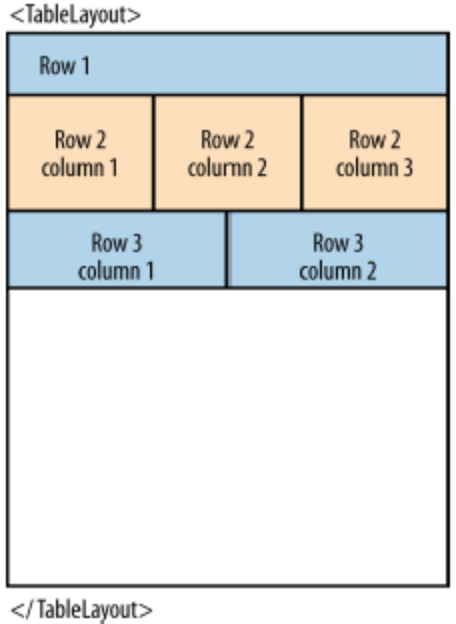
- Android organizes its UI elements into *views* and *layouts*. Everything you see, such as a button, label, or text box, is a view.
- Layouts organize views, such as grouping together a button and label or a group of these elements.
- Layouts can also be nested
- If you have prior experience with Java AWT or Swing, layouts are similar to Java containers and views are similar to Java components. Views in Android are sometimes referred to as *widgets*.

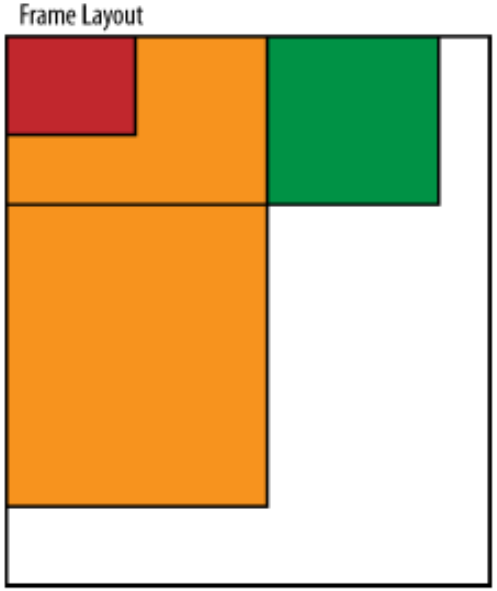
User Interfaces – Layouts

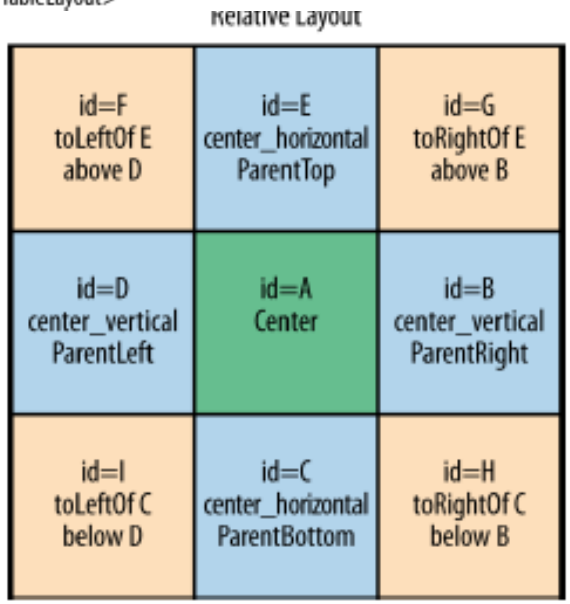
- There are a number of Layouts available in Android, Linear Layout, Table Layout, Frame Layout and Relative Layout
- Linear Layout simply lays out its children next to one another, either horizontally or vertically. The order Views are added matters, “older” views take precedence.
- TableLayout lays out its children in a table, and the views it contains are TableRow widgets. Each TableRow represents a row in a table and can contain other UI widgets. TableRows contain other Views laid out horizontally/vertically
- FrameLayout places its children on top of each other so that the latest child is covering the previous one, like a deck of cards. This layout policy is useful for tabs, as one example.
- RelativeLayout lays out its children relative to each other. It is very powerful because it doesn’t require you to nest extra layouts to achieve a certain look.
- RelativeLayout can minimize the total number of widgets that need to be drawn, thus improving the overall performance of your application.
- RelativeLayout adds a bit of complexity by requiring each child view to have an ID so that you can position it relative to other children, but its worth it.

User Interfaces – Layouts









Widget Properties

- There are lots of properties that can be set for widgets but the main ones you will use regularly are:

- layout_height *and* layout_width, define how much space this widget is asking from its parent layout to display itself. Because your application could run on many different devices with various screen sizes, you want to use relative sizes for your components, so the best practice is to use either match_parent or wrap_content for the value.
    - match_parent means that your widget wants all the available space from its parent, wrap_content means that it requires only as much space as it needs to display its own content.
- layout_weight, can be used to set the weighting or priority of widgets within a Layout
- layout_gravity, specifies how this particular widget is positioned within its parent layout, both horizontally and vertically. Values can be top, center, left, and so on.
- gravity, specifies how the content of this widget is positioned within the widget.
- text, some widgets with the text property include Button, EditText, and TextView. This property simply specifies the text to show in the widget. It is not a good practice to just enter the text, because then your layout will work in only one locale/language. Best practice is to define all text in the *strings.xml* resource file and refer to them from code.
- id, id is simply the unique identifier for this particular widget in a particular layout resource file. Not every widget needs an id but widgets that you need to manipulate later from Java do need ids. An id has the format @+id/*someName*

User Interface Components - Views

- User interface components or widgets in android are called Views. The following is a list of some of the more common ones.
- TextView: A standard read-only text label. It supports multiline display, string formatting, and automatic word wrapping.
- EditText: An editable text entry box. It accepts multiline entry, word-wrapping, and hint text.
- ListView: A View Group that creates and manages a vertical list of Views, displaying them as rows within the list.
- Spinner: A composite control that displays a Text View and an associated List View that lets you select an item from a list to display in the textbox.
- Button: A standard push-button.
- CheckBox: A two-state button represented by a checked or unchecked box.
- RadioButton: A two-state grouped button. A group of these presents the user with a number of binary options of which only one can be enabled at a time.
- QuickContactBadge: Displays a badge showing the image icon assigned to a contact you specify using a phone number, name, e-mail address, or URI.
- For more see, http://developer.android.com/guide/tutorials/views/index.html

Button

A clickable widget with a text label

- key attributes:

`android:clickable="`***`bool`***`"`	set to false to disable the button
`android:id="@+id/`***`theID`***`"`	unique ID for use in Java code
`android:onClick="`***`function`***`"`	function to call in activity when clicked (must be public, void, and take a View arg)
`android:text="`***`text`***`"`	text to put in the button

- represented by `Button` class in Java code

```
Button b = (Button) findViewById(R.id.theID);
```

ImageButton

A clickable widget with an image label

- key attributes:

`android:clickable="`***`bool`***`"`	set to false to disable the button
`android:id="@+id/`***`theID`***`"`	unique ID for use in Java code
`android:onClick="`***`function`***`"`	function to call in activity when clicked (must be public, void, and take a View arg)
`android:src="@drawable/`***`img`***`"`	image to put in the button (must correspond to an image resource)

- to set up an image resource:
  - put image file in project folder **app/src/main/res/drawable**
  - use `@drawable/foo` to refer to `foo.png`
    - use simple file names with only letters and numbers

EditText

An editable text input box

- key attributes:

`android:hint="`***`text`***`"`	gray text to show before user starts to type
`android:id="@+id/`***`theID`***`"`	unique ID for use in Java code
`android:inputType="`***`type`***`"`	what kind of input is being typed; `number`, `phone`, `date`, `time`, ...
`android:lines="`***`int`***`"`	number of visible lines (rows) of input
`android:maxLines="`***`int`***`"`	max lines to allow user to type in the box
`android:text="`***`text`***`"`	initial text to put in box (default empty)
`android:textSize="`***`size`***`"`	size of font to use (e.g. "20dp")

ImageView

Displays an image without being clickable

- key attributes:

`android:id="@+id/`***`theID`***`"`	unique ID for use in Java code
`android:src="@drawable/`***`img`***`"`	image to put in the screen (must correspond to an image resource)

- to change the visible image, in Java code:
  - get the ImageView using findViewById
  - call its **setImageResource** method and pass R.drawable.***filename***

RadioButton

A toggleable on/off switch; part of a group

- key attributes:

`android:checked="`***`bool`***`"`	set to true to make it initially checked
`android:clickable="`***`bool`***`"`	set to false to disable the button
`android:id="@+id/`***`theID`***`"`	unique ID for use in Java code
`android:onClick="`***`function`***`"`	function to call in activity when clicked (must be public, void, and take a View arg)
`android:text="`***`text`***`"`	text to put next to the button

- need to be nested inside a `RadioGroup` tag in XML so that only one can be selected at a time

RadioGroup in XML

```
<LinearLayout ...
        android:orientation="vertical"
        android:gravity="center|top">
    <RadioGroup ...
            android:orientation="horizontal">
        <RadioButton ... android:id="@+id/lions"
                        android:text="Lions"
                        android:onClick="radioClick" />
        <RadioButton ... android:id="@+id/tigers"
                        android:text="Tigers"
                        android:checked="true"
                        android:onClick="radioClick" />
        <RadioButton ... android:id="@+id/bears"
                        android:text="Bears, oh my!"
                        android:onClick="radioClick" />
    </RadioGroup>
</LinearLayout>
```

```
// in MainActivity.java
public class MainActivity extends Activity {

    public void radioClick(View view) {
        // check which radio button was clicked
        if (view.getId() == R.id.lions) {
            // ...
        } else if (view.getId() == R.id.tigers) {
            // ...
        } else {
            // bears ...
        }
    }
```

Layouts

How does the programmer specify where each component appears, how big each component should be, etc.?

- **Absolute positioning** (C++, C#, others):
  - Programmer specifies exact pixel coordinates of every component.
  - "Put this button at (x=15, y=75) and make it 70x31 px in size."

- **Layout managers** (Java, Android):
  - Objects that decide where to position each component based on some general rules or criteria.
    - "Put these four buttons into a 2x2 grid and put these text boxes in a horizontal flow in the south part of the app."
  - More flexible and general; works better with a variety of devices.

Layouts

- ViewGroup superclass represents containers of widgets/views
  - layouts are described in **XML** and mirrored in Java code
  - Android provides several pre-existing layout managers; you can define your own **custom layouts** if needed
  - layouts can be **nested** to achieve combinations of features

- in the Java code and XML:
  - an **Activity** is a ViewGroup
  - various Layout classes are also ViewGroups
  - widgets can be added to a ViewGroup, which will then manage that widget's position/size behavior

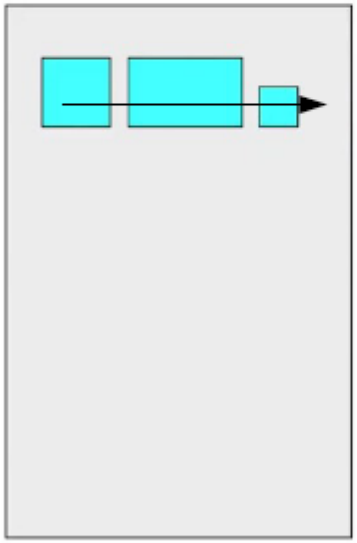
MainActivity.java × activity_main.xml ×

```
1  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2      xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
3      android:layout_height="match_parent" android:paddingLeft="16dp"
4      android:paddingRight="16dp"
5      android:paddingTop="16dp"
6      android:paddingBottom="16dp" tools:context=".MainActivity">
7
8  </LinearLayout>
9
```

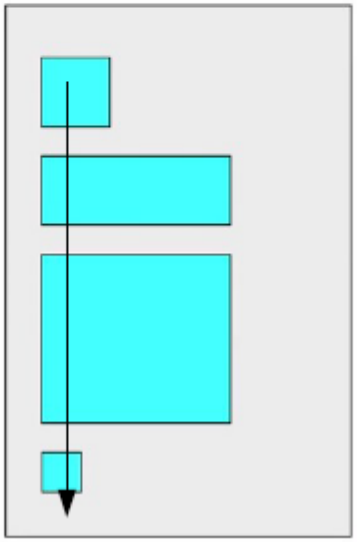
Linear Layouts

- lays out widgets/views in a single line
- **orientation** of horizontal (default) or vertical
- items do *not* wrap if they reach edge of screen!

horizontal



vertical



Linear Layouts – Eg1

```
<LinearLayout ...
        android:orientation="horizontal"
        tools:context=".MainActivity">
    <Button ... android:text="Button 1" />
    <Button ... android:text="Button 2 Hooray" />
    <Button ... android:text="Button 3" />
    <Button ... android:text="Button 4
                        Very Long Text" />
</LinearLayout>
```

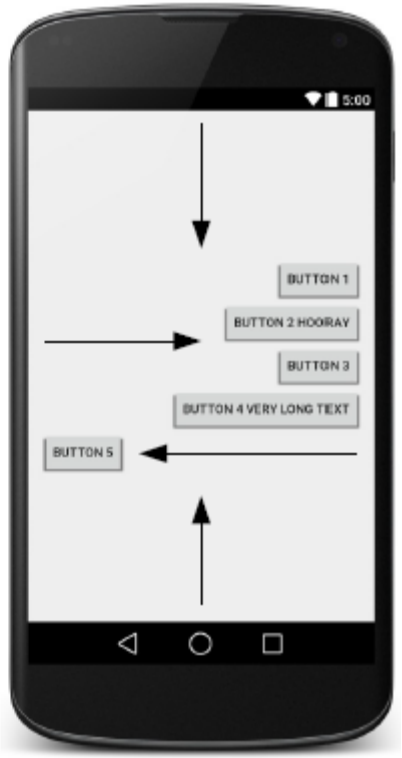
- In our examples, we'll use `...` when omitting boilerplate code that is auto-generated by Android Studio and not relevant to the specific example at hand.

Linear Layouts – Eg2

```
<LinearLayout ...
iew file attachments
        android:orientation="vertical"
        tools:context=".MainActivity">
    <Button ... android:text="Button 1" />
    <Button ... android:text="Button 2
                                Hooray" />
    <Button ... android:text="Button 3" />
    <Button ... android:text="Button 4
                          Very Long Text" />
</LinearLayout>
```

Layouts – Gravity

- **gravity**: alignment direction that widgets are pulled
  - top, bottom, left, right, center
  - combine multiple with |
  - set `gravity` on the layout to adjust all widgets; set `layout_gravity` on an individual widget




```
<LinearLayout ...
        android:orientation="vertical"
        android:gravity="center|right">
    <Button ... android:text="Button 1" />
    <Button ... android:text="Button 2 Hooray" />
    <Button ... android:text="Button 3" />
    <Button ... android:text="Button 4 Very Long Text" />
    <Button ... android:text="Button 5"
                android:layout_gravity="left" />
</LinearLayout>
```

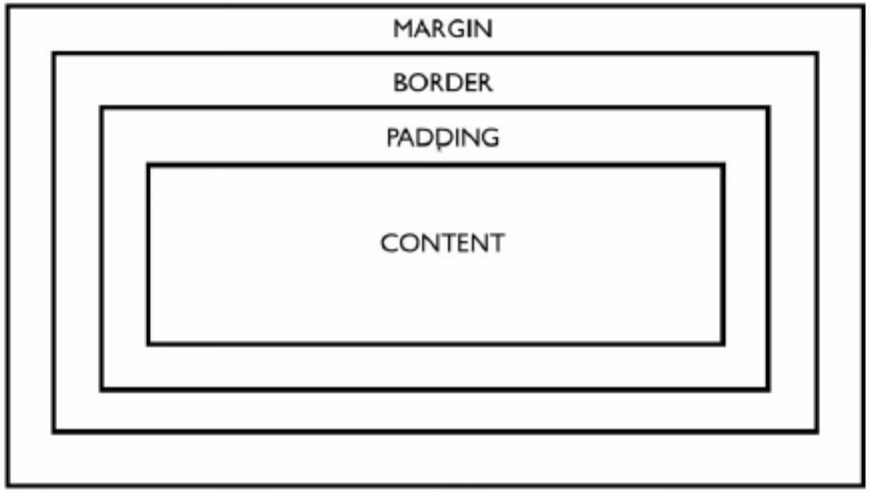
Layouts – Weight

- **weight**: gives elements relative sizes by integers
  - widget with weight ***K*** gets ***K***/total fraction of total size
  - cooking analogy: "2 parts flour, 1 part water, ..."

```
<LinearLayout ...
        android:orientation="vertical">
    <Button ... android:text="B1"
              android:layout_weight="1" />
    <Button ... android:text="B2"
              android:layout_weight="3" />
    <Button ... android:text="B3"
              android:layout_weight="1" />
</LinearLayout>
```

Weight Box Model

- **content**: every widget or view has a certain size (width x height) for its content, the widget itself
- **padding**: you can artificially increase the widget's size by applying padding in the widget just outside its content
- **border**: outside the padding, a line around edge of widget
- **margin**: separation from neighboring widgets on screen



Sizing a Widget

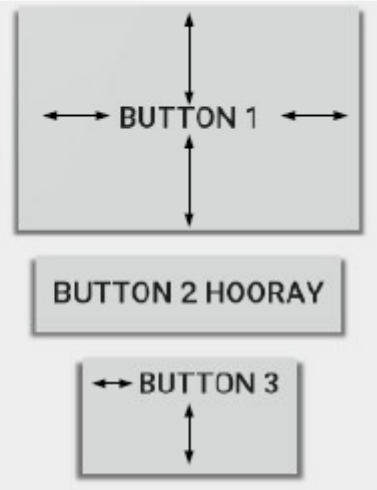
- **width** and **height** of a widget can be:
  - `wrap_content` : exactly large enough to fit the widget's content
  - `match_parent` : as wide or tall as 100% of the screen or layout
  - a specific fixed width such as 64dp *(not usually recommended)*
    - *dp = device pixels; dip = device-independent pixels; sp = scaling pixels*

```
<Button ...
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
```

Padding

- **padding**: extra space *inside* widget
  - set padding to adjust all sides; paddingTop, Bottom, Left, Right for one side
  - usually set to specific values like 10dp

    (some widgets have a default value ~16dp)

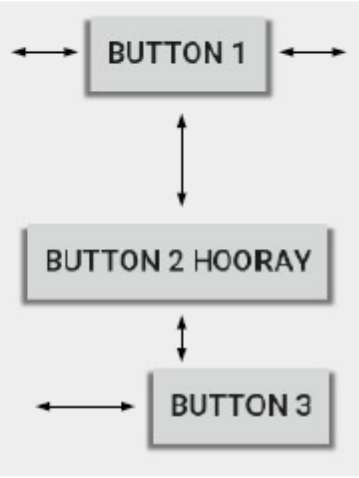



```
<LinearLayout ...
        android:orientation="vertical">
    <Button ... android:text="Button 1"
              android:padding="50dp" />
    <Button ... android:text="Button 2 Hooray" />
    <Button ... android:text="Button 3"
              android:paddingLeft="30dp"
              android:paddingBottom="40dp" />
</LinearLayout>
```

Margins

- **margin**: extra space *outside* widget to separate it from others
  - set `layout_margin` to adjust all sides; layout_marginTop, Bottom, Left, Right
  - usually set to specific values like 10dp

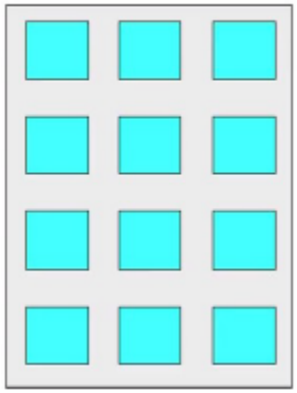
  *(set defaults in res/values/**dimens.xml**)*




```
<LinearLayout ...
        android:orientation="vertical">
    <Button ... android:text="Button 1"
              android:layout_margin="50dp" />
    <Button ... android:text="Button 2 Hooray" />
    <Button ... android:text="Button 3"
              android:layout_marginLeft="30dp"
              android:layout_marginTop="40dp" />
</LinearLayout>
```

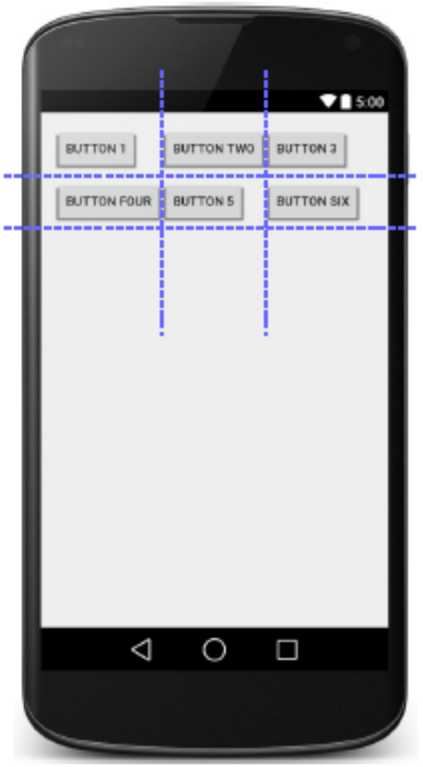
GridLayout

- lays out widgets/views in lines of **rows** and **columns**
  - orientation attribute defines row-major or column-major order
  - introduced in Android 4; replaces older TableLayout
- by default, rows and columns are equal in size
  - each widget is placed into "next" available row/column index unless it is given an explicit `layout_row` and `layout_column` attribute
  - grid of 4 rows, 3 columns:



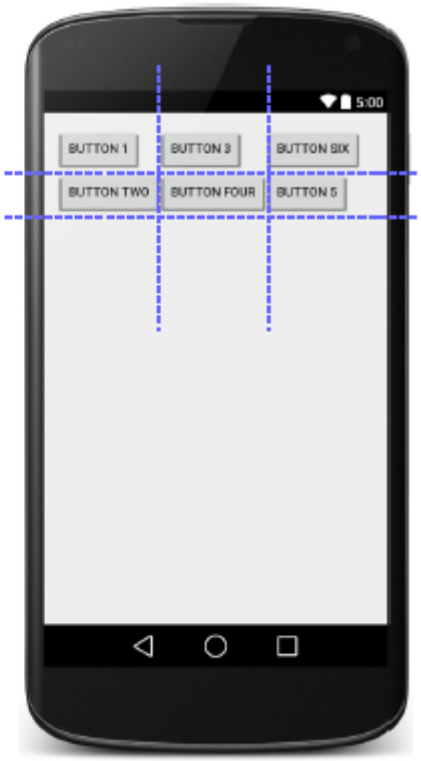
GridLayout Example

```
<GridLayout ...
        android:rowCount="2"
        android:columnCount="3"
        tools:context=".MainActivity">
    <Button ... android:text="Button 1" />
    <Button ... android:text="Button Two" />
    <Button ... android:text="Button 3" />
    <Button ... android:text="Button Four" />
    <Button ... android:text="Button 5" />
    <Button ... android:text="Button Six" />
</GridLayout>
```



GridLayout Example 2

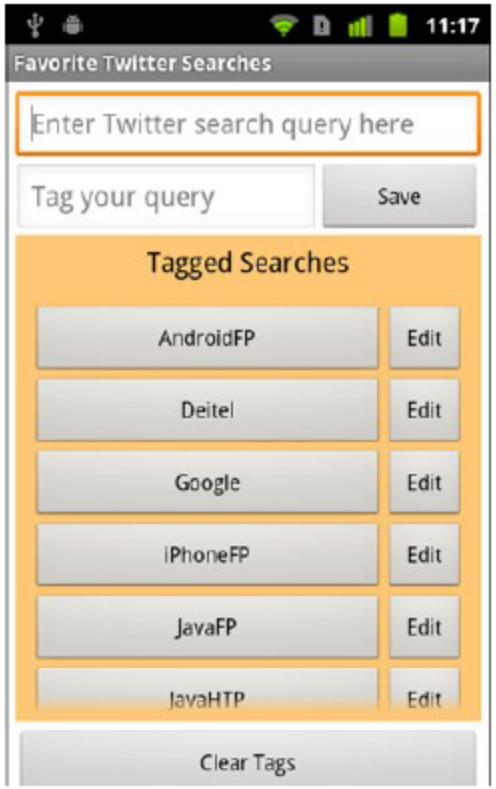
```
<GridLayout ...
        android:rowCount="2"
        android:columnCount="3"
        android:orientation="vertical">
    <Button ... android:text="Button 1" />
    <Button ... android:text="Button Two" />
    <Button ... android:text="Button 3" />
    <Button ... android:text="Button Four" />
    <Button ... android:text="Button 5"
                android:layout_row="1"
                android:layout_column="2" />
    <Button ... android:text="Button Six"
                android:layout_row="0"
                android:layout_column="2" />
```



Nested Layout

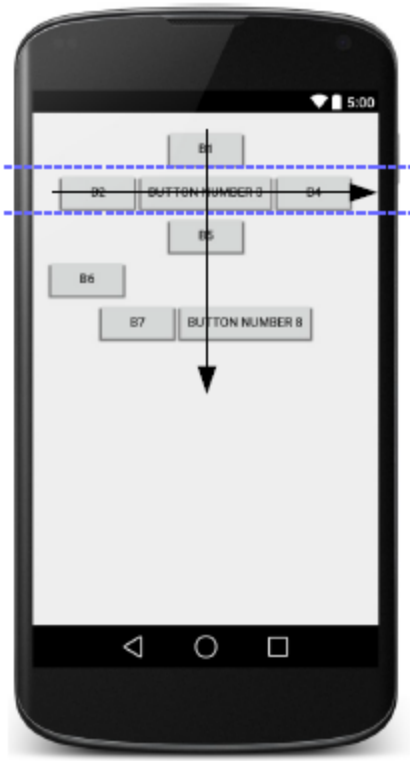
- to produce more complicated appearance, use a **nested** layout
  - (layouts inside layouts)

```
<OuterLayoutType ...>

    <InnerLayoutType ...>
        <Widget ... />
        <Widget ... />
    </InnerLayoutType>

    <InnerLayoutType ...>
        <Widget ... />
        <Widget ... />
    </InnerLayoutType>

    <Widget ... />
    <Widget ... />
</OuterLayoutType>
```



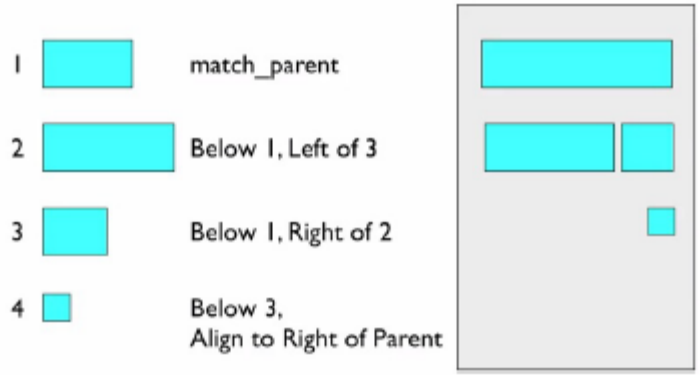
Nested Layout Example

```
<LinearLayout ...
        android:orientation="vertical" android:gravity="center|top">
    <Button ... android:text="B1" />
    <LinearLayout ...
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="horizontal"
            android:gravity="center|top">
        <Button ... android:text="B2" />
        <Button ... android:text="Button Number 3" />
        <Button ... android:text="B4" />
    </LinearLayout>
    <Button ... android:text="B5" />
    <Button ... android:text="B6" android:layout_gravity="left" />
    <LinearLayout ...
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="horizontal"
            android:gravity="center|top">
        <Button ... android:text="B7" />
        <Button ... android:text="Button Number 8" />
    </LinearLayout>
</LinearLayout>
```



Relative Layout

- each widget's position and size are relative to other views
  - relative to "parent" (the activity itself)
  - relative to other widgets/views
  - x-positions of reference: left, right, center
  - y-positions of reference: top, bottom, center
- intended to reduce the need for nested layouts



Relative Layout – Anchor Points

- properties for x/y relative to **another widget**:
  - **layout_**below, above, toLeftOf, toRightOf
    - set these to the ID of another widget in the format "@id/***theID***"
      (obviously, the given widget must have an ID for this to work)

- properties for x/y relative to layout **container** (the activity):
  - **layout_alignParent**Top, Bottom, Left, Right
    - set these flags to a boolean value of "true" to enable them
  - **layout_center**Horizontal, Vertical, InParent
    - set these flags to "true" to center the control within its parent in a dimension

Relative Layout – Example

```
<RelativeLayout ... >
    <Button ... android:id="@+id/b1" android:text="B1"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" />
    <Button ... android:id="@+id/b2" android:text="B2"
        android:layout_alignParentLeft="true"
        android:layout_below="@+id/b1" />
    <Button ... android:id="@+id/b3" android:text="B3"
        android:layout_centerHorizontal="true"
        android:layout_below="@+id/b2" />
    <Button ... android:id="@+id/b4" android:text="B4"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/b2" />
    <TextView ... android:id="@+id/tv1"
        android:text="I'm a TextView!"
        android:layout_centerInParent="true" />
    <Button ... android:id="@+id/b5" android:text="B5"
        android:padding="50dp"
        android:layout_centerHorizontal="true"
        android:layout_alignParentBottom="true"
        android:layout_marginBottom="50dp" />
</RelativeLayout>
```

Frame Layout

- meant to hold only a single widget inside, which occupies the entirety of the activity
  - most commonly used with layout fragments (seen later)
  - less useful for more complex layouts

    (can put in multiple items and move them to "front" in Z-order)

```
<FrameLayout ... >
    <ImageView
     android:src="@drawable/jellybean"
     ... />
</FrameLayout>
```

Menus

- Pop up menus are useful in Mobile Apps because they are launched with the press of a button (hard/soft) and are not persistent and thus no occupying valuable screen space.

- The XML for the app menu will be in the menu subfolder of the resources folder. One is created by default with one item in it. The menu can be launched but is a **dummy menu** until its populated by the developer. The default menu XML looks something like this:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/menu_settings"
        android:orderInCategory="100"
        android:showAsAction="never"
        android:title="@string/menu_settings"/>

</menu>
```

- The first **android:id** parameter names this menu item **menu_settings** so that we can reference it in our Java code. The second **android:orderInCategory** parameter sets the order that the menu item will be in a menu.

- The third **android:showAsAction** parameter determines whether your menu shows on the Action Icon Bar on Android OS version 3.x and 4.x.

- The final **android:title** parameter is the title, or label, for the menu item within the pop-up menu itself. Because this menu title is a string constant, it is set in the **values** folder in the **strings.xml** file

Menus, Inflating a menu

- We don’t need to write the Menu Inflater code to create the menu, because the New Android Application Project helper did that for us, however, let’s take a closer look at it

```
public boolean onCreateOptionsMenu(Menu menu) {

getMenuInflater().inflate(R.menu.activity_main, menu);

return true;

}
```

- The onCreateOptionsMenu( ) method is part of the Android Activity class, and it is used to create the options menu for any given Activity.

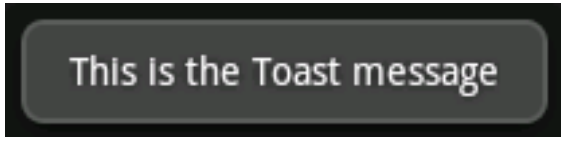
- This method is passed a Menu object named menu into the method, where it is populated with your menu XML parameters via an inflate( ) method, which references your \resource\menu\activity_main.xml file via the first parameter in the method, R.menu.activity_main

Displaying Toasts

- A "Toast" is a pop-up message that appears for a short time.
- Useful for displaying short updates in response to events.
- Should not be relied upon extensively for important info.

Toast.makeText(this,"message",duration).show();

– where duration is Toast.LENGTH_SHORT or LENGTH_LONG



Intents and Events

• An Intent is a messaging object you can use to request an action from another app component. Although intents facilitate communication between components in several ways, there are three fundamental use-cases:

1. You can start a new instance of an Activity by passing an Intent to startActivity(). The Intent describes the activity to start and carries any necessary data.
2. You can start a service to perform a one-time operation (such as download a file) by passing an Intent to startService(). The Intent describes the service to start and carries any necessary data.
3. To deliver a broadcast, a broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. You can deliver a broadcast to other apps by passing an Intent to sendBroadcast()

Intents Types

- There are two types of intents: Explicit and Implicit Intents
- Explicit intents specify the component to start by name (the fully-qualified class name). You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start. For example, start a new activity in response to a user action or start a service to download a file in the background.
- Implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.
- When you create an explicit intent to start an activity or service, the system immediately starts the app component specified in the Intent object.

Implicit Intents

- When you create an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the intent filters declared in the manifest file of other apps on the device.
- If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

- An intent filter is an expression in an app's manifest file that specifies the type of intent that the component would like to receive. For instance, by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with a certain kind of intent. Likewise, if you do not declare any intent filters for an activity, then it can be started only with an explicit intent.

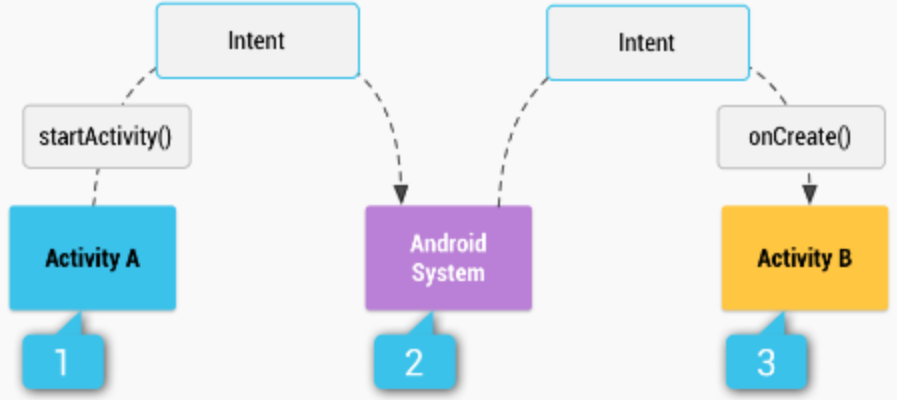


Figure 1. Illustration of how an implicit intent is delivered through the system to start another activity: **[1]** *Activity A* creates an `Intent` with an action description and passes it to `startActivity()`. **[2]** The Android System searches all apps for an intent filter that matches the intent. When a match is found, **[3]** the system starts the matching activity (*Activity B*) by invoking its `onCreate()` method and passing it the `Intent`.

Building an Intent Object

• An Intent object carries information that the Android system uses to determine which component to start (such as the exact component name or component category that should receive the intent), plus information that the recipient component uses in order to properly perform the action (such as the action to take and the data to act upon).The Intent object will send the following kind of information to the components

- **component name** of the component (class) which needs to process the intent
- The **action** that needs to be performed
- The **data** that the action needs to operate on
- The **type** of data (a MIME type) that is being processed
- The **category** that this processing falls under
- Any **flags** and **extras** that are needed to further define this processing that needs to be performed.

Example Explicit Intent

- If you built a service in your app, named DownloadService, designed to download a file from the web, you can start it with the following code:

```
// Executed in an Activity, so 'this' is the Context

//The fileUrl is a string URL, such as
//http://www.example.com/image.png

Intent downloadIntent = new Intent(this,DownloadService.class);

downloadIntent.setData(Uri.parse(fileUrl));

startService(downloadIntent);
```

- The Intent(Context, Class) constructor supplies the app Context and the component a Class object. As such, this intent explicitly starts the DownloadService class in the app.

Example Implicit Intent

- Using an implicit intent is useful when your app cannot perform the action, but other apps probably can and you'd like the user to pick which app to use.

- For example, if you have content you want the user to share with other people, create an intent with the ACTION_SEND action and add extras that specify the content to share. When you call startActivity() with that intent, the user can pick an app through which to share the content.

- Caution: It's possible that a user won't have any apps that handle the implicit intent you send to startActivity(). If that happens, the call will fail and your app will crash. To verify that an activity will receive the intent, call resolveActivity() on your Intent object. If the result is non-null, then there is at least one app that can handle the intent and it's safe to call startActivity(). If the result is null, you should not use the intent.

```
// Create the text message with a string
Intent sendIntent = newIntent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType(HTTP.PLAIN_TEXT_TYPE); // "text/plain" MIME type
// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
startActivity(sendIntent);
}
```

Events

- **Events** allow communication at a much more localised level within an Android application, as they enable the lowest level component parts of an app, which are usually the user interface elements, or **widgets**, to talk, or more accurately to **call back**, to your Java programming logic.

- Imagine a scenario where we want our app activity to launch another activity through an Intent object. This will be triggered by the user through a button press or a menu selection. We need to register for that event and handle it by doing something (launching the intent).

- You should be fairly familiar with this process as you have seen it before with standard Java, the code will be something like this:

```
Button someButton = (Button)findViewById(R.id.someButton);
someButton.setOnClickListener(new OnClickListener() {
    public void onClick(View v) { //Handle the Event} } );
```

- If we wanted to return to the Main activity from the one launched by the Intent we can call the finish() method in from the Activity class, e.g. we could call this in the event handler of a “Done” button of the spawned Activity.

Events – Key Events

- While click events are the most common events in Android there are also key events that are generated from keyboards or keypads. Lets consider the onKeyDown() event handler from the KeyEvent class that is called when a key is pressed down

- The onKeyDown(int keyCode, Event KeyEvent) method takes in two parameters, one integer and one object. The first parameter contains the keyCode, which is an integer value that represents the numeric value (constant) for the key that is being depressed. The second parameter is a KeyEvent object named event that is the key event that is being handled.

- Here is a sample for detecting that the ‘x’ key has been pressed to exit the activity, note the constant KeyEvent.KEYCODE_X for detecting an ‘x’ depression

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_X) {
                finish();
                return true;
        }
        return false;
}
```

Events – Other Events

- There are other events such as the onLongClick( ) event handling method, which is the Android equivalent of the right-click on a computer or other device using a mouse.
- A LongClick is invoked by touching and holding the screen, trackball, or the center selection button on any Android device. A onLongClick() event handler is implemented in your Java code just like an onClick( ) event handler is.
- The onCreateContextMenu( ) event handler method handles context menu events. Context Menus are also similar to the menus found in a PC O/S that are accessed via right-clicking an object or area of the GUI to get a menu of Context-Sensitive options that are available specifically for that object.
- The onFocusChange( ) event handler method is used for handling focus events that are sent out by user interface elements when the user progresses from one element to the next.