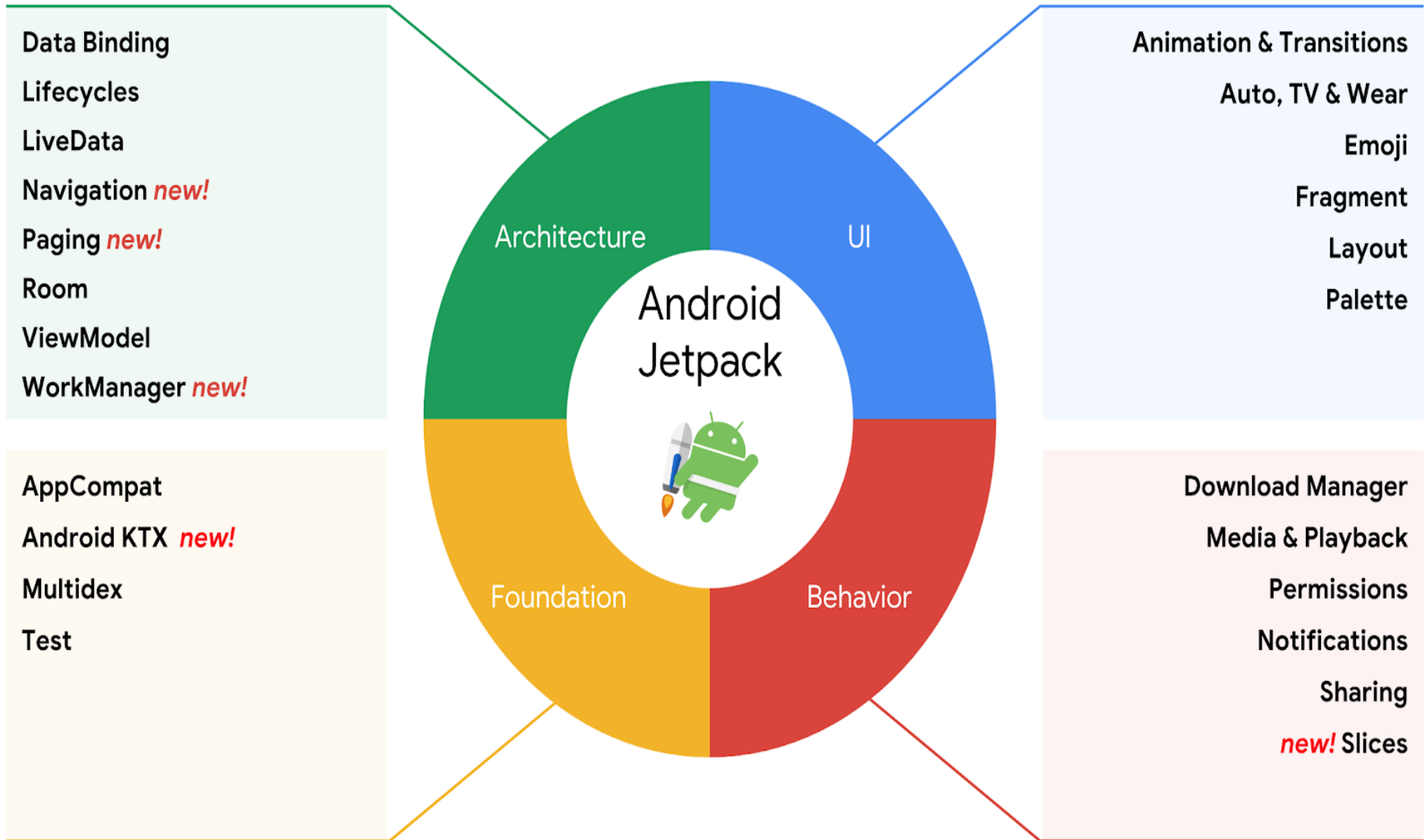




Android Jetpack

<https://developer.android.com/topic/libraries/architecture/>

Components



Android architecture components




- Android architecture components are a collection of libraries that help you design robust, testable, and maintainable apps.
- Start with classes for managing your UI component lifecycle and handling data persistence.
 - Manage your app's lifecycle with ease. New lifecycle-aware components help you manage your activity and fragment lifecycles. Survive configuration changes, avoid memory leaks and easily load data into your UI.
 - Use LiveData to build data objects that notify views when the underlying database changes.
 - ViewModel Stores UI-related data that isn't destroyed on app rotations.
 - Room is an a SQLite object mapping library. Use it to Avoid boilerplate code and easily convert SQLite table data to Java objects. Room provides compile time checks of SQLite statements and can return RxJava, Flowable and LiveData observables.

Introduction to Activities



- The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.
- Unlike programming paradigms in which apps are launched with a `main()` method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

- 
- The mobile-app experience differs from its desktop counterpart in that a user's interaction with the app doesn't always begin in the same place.
 - Instead, the user journey often begins non-deterministically.
 - For instance, if you open an email app from your home screen, you might see a list of emails.
 - By contrast, if you are using a social media app that then launches your email app, you might go directly to the email app's screen for composing an email.

Cont ...



- An activity provides the window in which the app draws its UI.
- This window typically fills the screen, but may be smaller than the screen and float on top of other windows.
- Generally, one activity implements one screen in an app.
- Each activity can then start another activity in order to perform different actions.

Cont ...

- Although activities work together to form a cohesive user experience in an app, each activity is only loosely bound to the other activities; there are usually minimal dependencies among the activities in an app.



Declare activities



- To declare your activity, open your manifest file and add an `<activity>` element as a child of the `<application>` element. For example:

```
<manifest ... >
```

```
<application ... >
```

```
    <activity android:name=".ExampleActivity" />
```

```
</application ... >
```

```
</manifest >
```

- The only required attribute for this element is `android:name`, which specifies the class name of the activity.

Declare intent filters



- Intent filters are a very powerful feature of the Android platform.
- They provide the ability to launch an activity based not only on an explicit request, but also an implicit one.
- For example, an explicit request might tell the system to "Start the Send Email activity in the Gmail app".
- By contrast, an implicit request tells the system to "Start a Send Email screen in any activity that can do the job."
- When the system UI asks a user which app to use in performing a task, that's an intent filter at work.

Cont ...



- You can take advantage of this feature by declaring an `<intent-filter>` attribute in the `<activity>` element.
- The definition of this element includes an `<action>` element and, optionally, a `<category>` element and/or a `<data>` element.
- These elements combine to specify the type of intent to which your activity can respond.
- For example, the following code snippet shows how to configure an activity that sends text data, and receives requests from other activities to do so:

Cont ...



```
<activity android:name=".ExampleActivity"  
android:icon="@drawable/app_icon">
```

```
<intent-filter>
```

```
    <action  
    android:name="android.intent.action.SEND" />
```

```
    <category  
    android:name="android.intent.category.DEFAULT" />
```

```
        <data android:mimeType="text/plain" />
```

```
    </intent-filter>
```

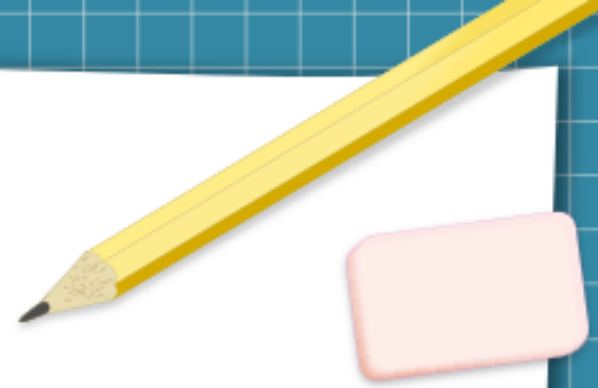
```
</activity>
```

Cont ...



- In this example, the `<action>` element specifies that this activity sends data.
- Declaring the `<category>` element as `DEFAULT` enables the activity to receive launch requests.
- The `<data>` element specifies the type of data that this activity can send.

Cont ...



```
Intent sendIntent = new Intent();  
sendIntent.setAction(Intent.ACTION_SEND);  
sendIntent.setType("text/plain");  
sendIntent.putExtra(Intent.EXTRA_TEXT,  
    textMessage);  
// Start the activity  
startActivity(sendIntent);
```

Declare permissions



- You can use the manifest's `<activity>` tag to control which apps can start a particular activity.
- A parent activity cannot launch a child activity unless both activities have the same permissions in their manifest.
- If you declare a `<uses-permission>` element for a particular activity, the calling activity must have a matching `<uses-permission>` element.

Cont ...



- For example, if your app wants to use a hypothetical app named SocialApp to share a post on social media, SocialApp itself must define the permission that an app calling it must have:

```
<manifest>
```

```
<activity android:name="...."  
android:permission="com.google.socialapp.per  
mission.SHARE_POST" />
```

Cont ...



- Then, to be allowed to call SocialApp, your app must match the permission set in SocialApp's manifest:

```
<manifest>
```

```
  <uses-permission  
    android:name="com.google.socialapp.permission.  
    n.SHARE_POST" />
```

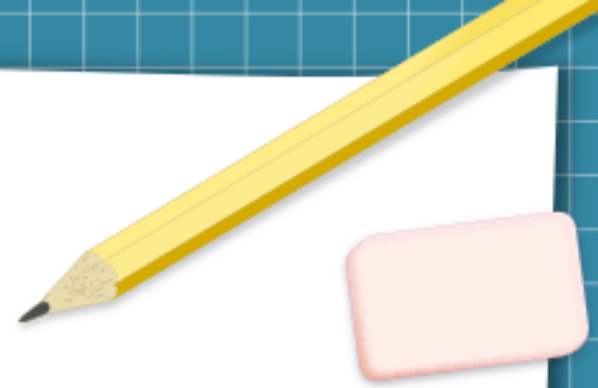
```
</manifest>
```


Managing the activity lifecycle



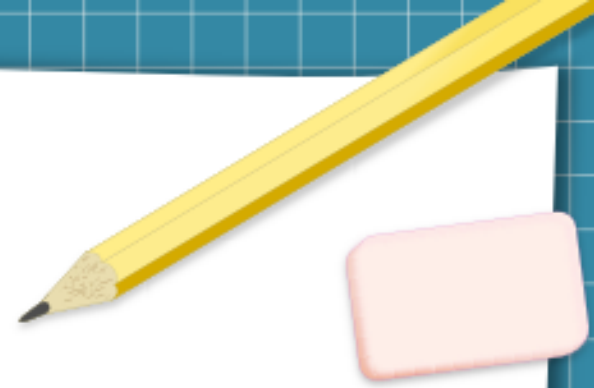
- Over the course of its lifetime, an activity goes through a number of states. You use a series of callbacks to handle transitions between states.
 - onCreate()
 - You must implement this callback, which fires when the system creates your activity.
 - Your implementation should initialize the essential components of your activity: For example, your app should create views and bind data to lists here.
 - Most importantly, this is where you must call setContentView() to define the layout for the activity's user interface.
 - When onCreate() finishes, the next callback is always onStart().

Cont ...



- **OnStart()**
 - As `onCreate()` exits, the activity enters the Started state, and the activity becomes visible to the user.
 - This callback contains what amounts to the activity's final preparations for coming to the foreground and becoming interactive.
- **OnResume()**
 - The system invokes this callback just before the activity starts interacting with the user.
 - At this point, the activity is at the top of the activity stack, and captures all user input.
 - Most of an app's core functionality is implemented in the `onResume()` method.
 - The `onPause()` callback always follows `onResume()`.

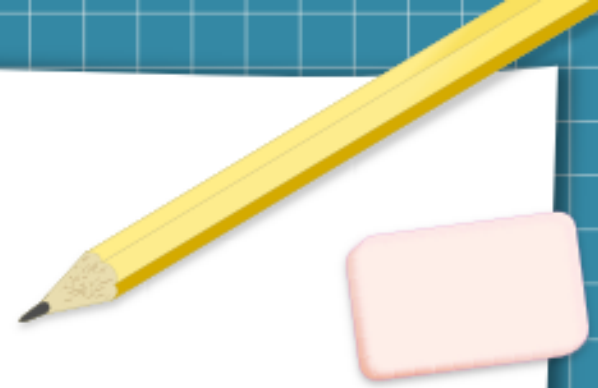
Cont ...



– onPause()

- The system calls onPause() when the activity loses focus and enters a Paused state.
- This state occurs when, for example, the user taps the Back or Recents button.
- When the system calls onPause() for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.
- An activity in the Paused state may continue to update the UI if the user is expecting the UI to update.
- **You should not use onPause() to save application or user data, make network calls, or execute database transactions.**
- Once onPause() finishes executing, the next callback is either onStop() or onResume(), depending on what happens after the activity enters the Paused state.

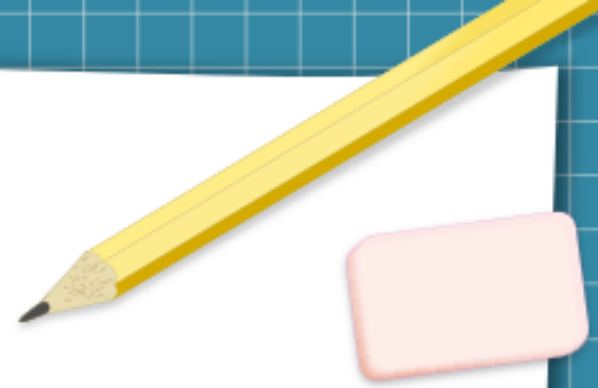
Cont ...



– OnStop()

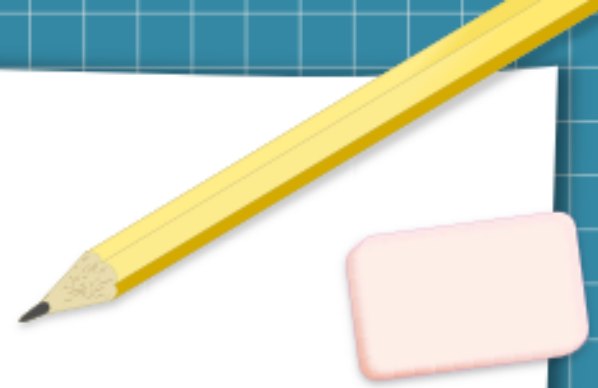
- The system calls `onStop()` when the activity is no longer visible to the user.
- This may happen because the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity.
- In all of these cases, the stopped activity is no longer visible at all.
- The next callback that the system calls is either `onRestart()`, if the activity is coming back to interact with the user, or by `onDestroy()` if this activity is completely terminating.

Cont ...



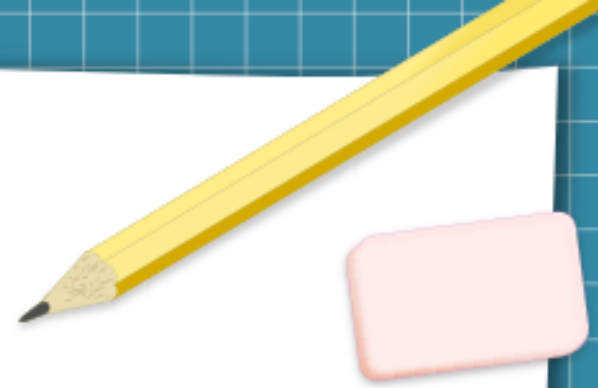
- OnRestart()
 - The system invokes this callback when an activity in the Stopped state is about to restart. onRestart() restores the state of the activity from the time that it was stopped.
 - This callback is always followed by onStart().
- OnDestroy()
 - The system invokes this callback before an activity is destroyed.
 - This callback is the final one that the activity receives. onDestroy() is usually implemented to ensure that all of an activity's resources are released when the activity, or the process containing it, is destroyed.

Cont ...



- Layouts (
<https://developer.android.com/guide/topics/ui/declaring-layout>
)
 - A layout defines the structure for a user interface in your app, such as in an activity.
 - All elements in the layout are built using a hierarchy of View and ViewGroup objects.
 - A View usually draws something the user can see and interact with.
 - Whereas a ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup objects

Cont ...



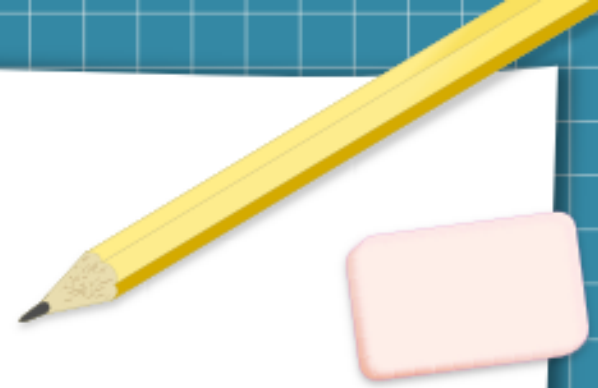
- The View objects are usually called "widgets" and can be one of many subclasses, such as Button or TextView.
- The ViewGroup objects are usually called "layouts" can be one of many types that provide a different layout structure, such as LinearLayout or ConstraintLayout
- You can declare a layout in two ways:
 - Declare UI elements in XML.
 - Instantiate layout elements at runtime

Load the XML Resource



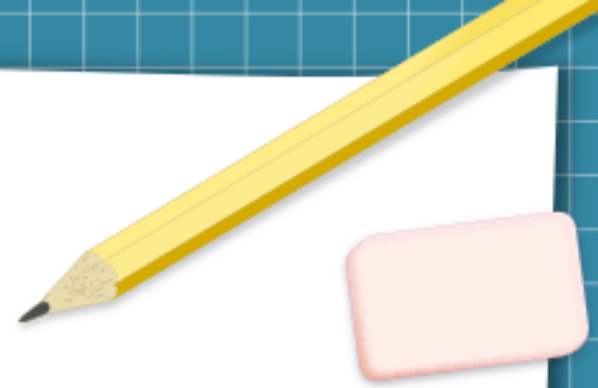
- When you compile your app, each XML layout file is compiled into a View resource.
- You should load the layout resource from your app code, in your `Activity.onCreate()` callback implementation.
- Do so by calling `setContentView()`, passing it the reference to your layout resource in the form of: `R.layout.layout_file_name`

Attributes



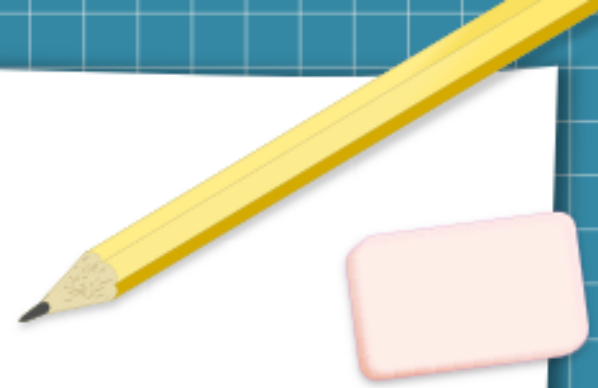
- Every View and ViewGroup object supports their own variety of XML attributes.
- Some attributes are specific to a View object (for example, TextView supports the textSize attribute), but these attributes are also inherited by any View objects that may extend this class.
- Some are common to all View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

Cont ...



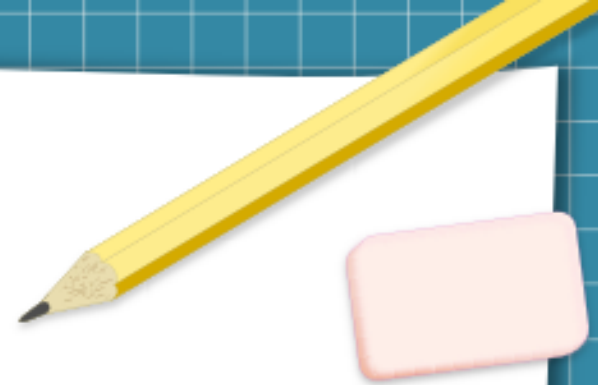
- And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

ID



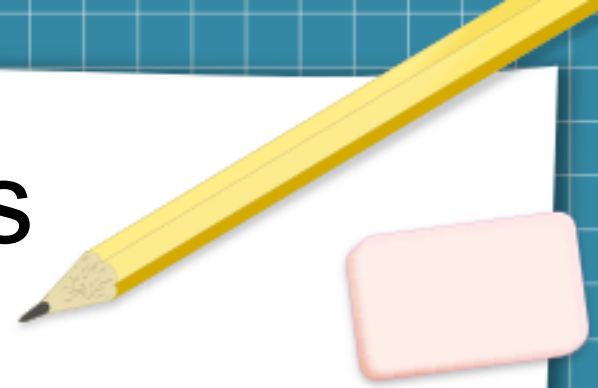
- Any View object may have an integer ID associated with it, to uniquely identify the View within the tree.
- When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the id attribute.
- This is an XML attribute common to all View objects (defined by the View class) and you will use it very often.
- The syntax for an ID, inside an XML tag is:
 - `android:id="@+id/my_button"`

Cont ...



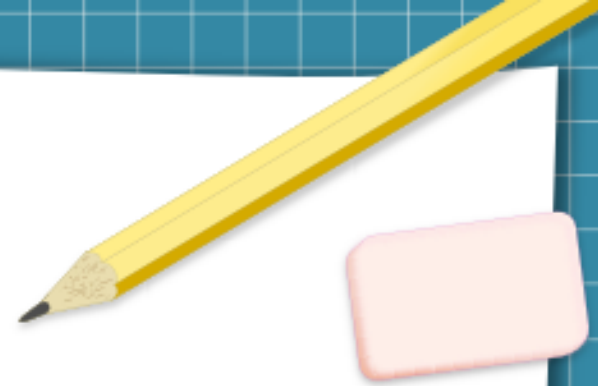
- The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource.
- The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the R.java file).
- There are a number of other ID resources that are offered by the Android framework.
- When referencing an Android resource ID, you do not need the plus-symbol, but must add the android package namespace, like so:
 - `android:id="@android:id/empty"`

Layout Parameters



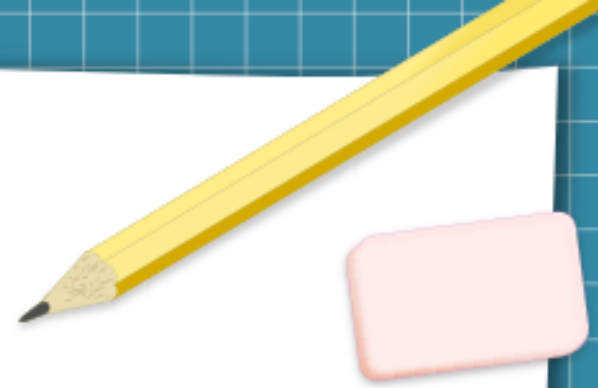
- XML layout attributes named `layout_something` define layout parameters for the View that are appropriate for the ViewGroup in which it resides.
- Every ViewGroup class implements a nested class that extends `ViewGroup.LayoutParams`.
- This subclass contains property types that define the size and position for each child view, as appropriate for the view group.

Layout Position

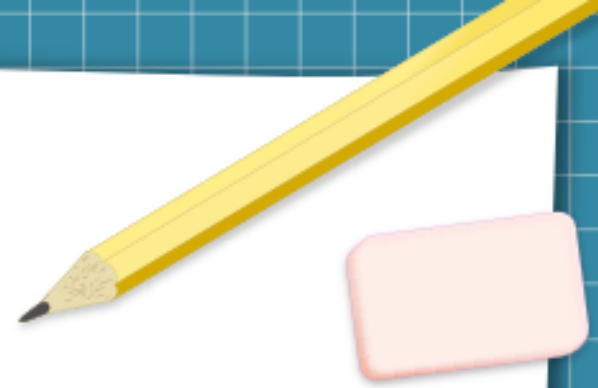


- The geometry of a view is that of a rectangle.
- A view has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height.
- The unit for location and dimensions is the pixel.

Common Layouts

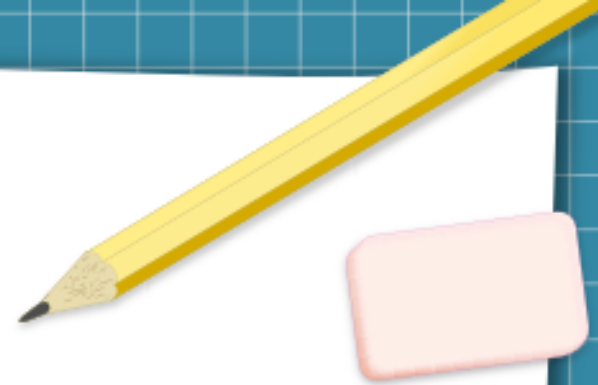


Linear Layout



- A layout that organizes its children into a single horizontal or vertical row.
- It creates a scrollbar if the length of the window exceeds the length of the screen.

Relative Layout

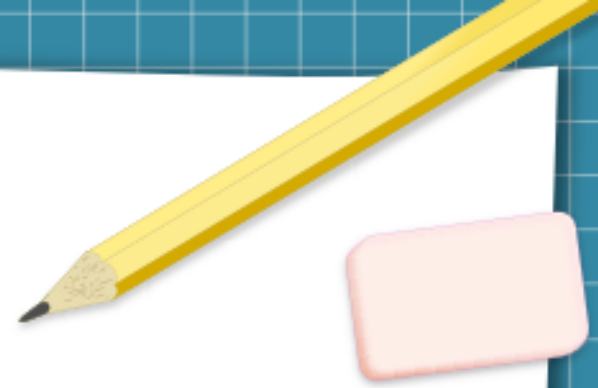


- Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

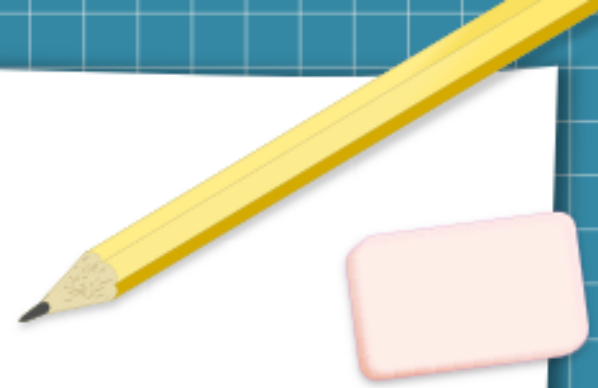
-

Web View

- Displays web pages.

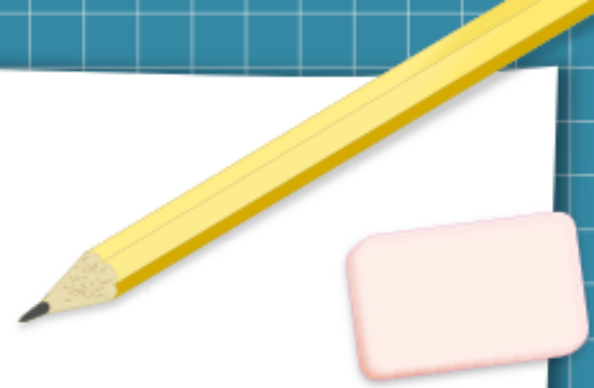


ConstraintLayout



- Is a ViewGroup which allows you to position and size widgets in a flexible way.
- Note: ConstraintLayout is available as a support library that you can use on Android systems starting with API level 9 (Gingerbread).
- There are currently various types of constraints that you can use:

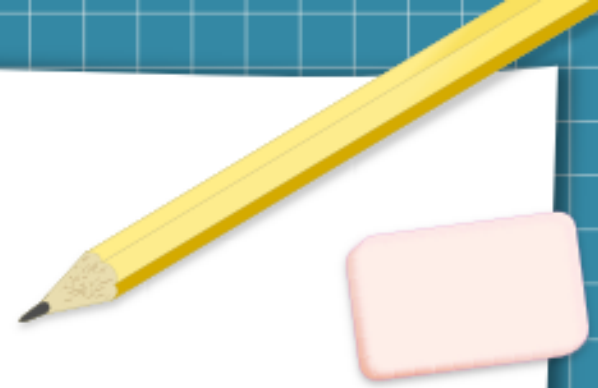
Cont ...



- Relative positioning
 - Allow you to position a given widget relative to another one.
 - You can constrain a widget on the horizontal and vertical axis:
- Margins
 - If side margins are set, they will be applied to the corresponding constraints, enforcing the margin as a space between the target and the source side.
- Centering positioning
- Circular positioning
 - You can constrain a widget center relative to another widget center, at an angle and a distance. This allows you to position a widget on a circle

Cont ...

- Visibility behavior
- Dimension constraints
- Chains
- Virtual Helpers objects
- Optimizer



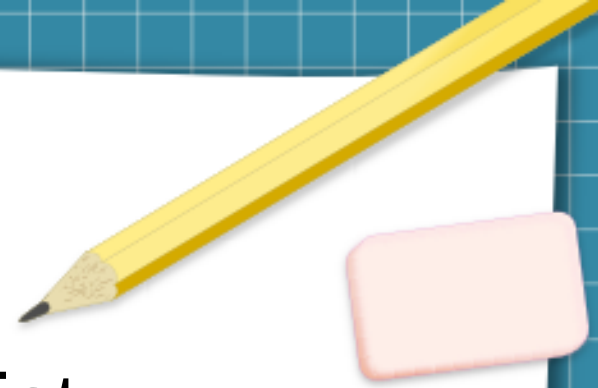
Layouts with an Adapter



- When the content for your layout is dynamic or not pre-determined, you can use 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.

List View

- Displays a scrolling single column list.



Grid View

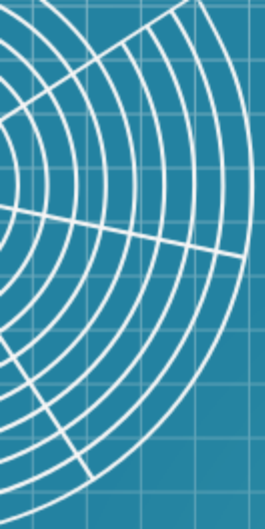
- Displays a scrolling grid of columns and rows



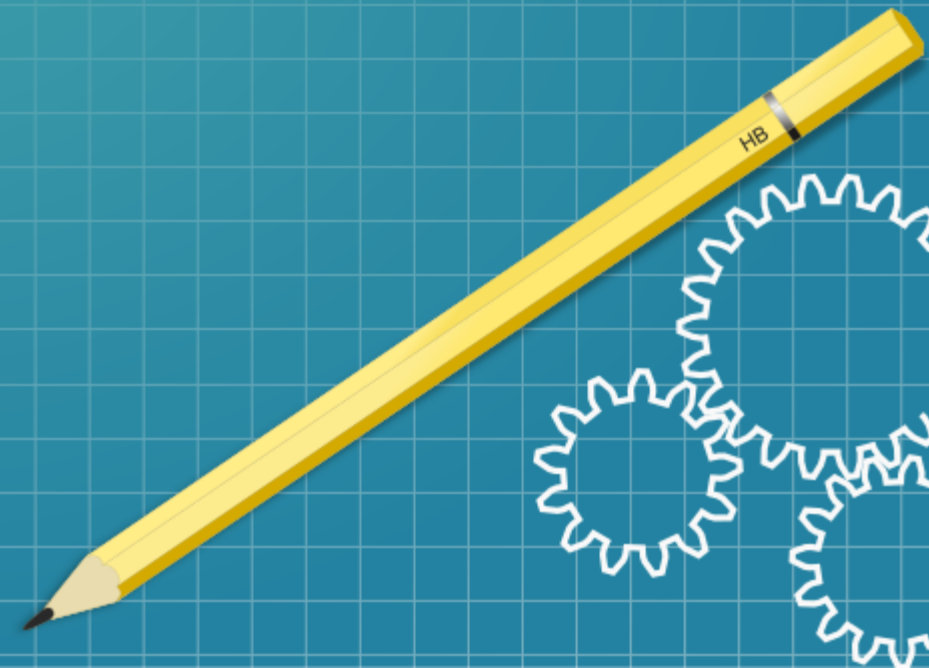
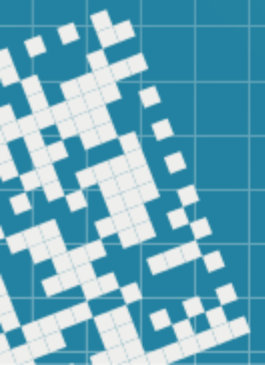
Filling an adapter view with data



- You can populate an AdapterView such as ListView or GridView by binding the AdapterView instance to an Adapter, which retrieves data from an external source and creates a View that represents each data entry.
- Android provides several subclasses of Adapter that are useful for retrieving different kinds of data and building views for an AdapterView. The two most common adapters are:
 - ArrayAdapter
 - SimpleCursorAdapter



TU-K SCIT

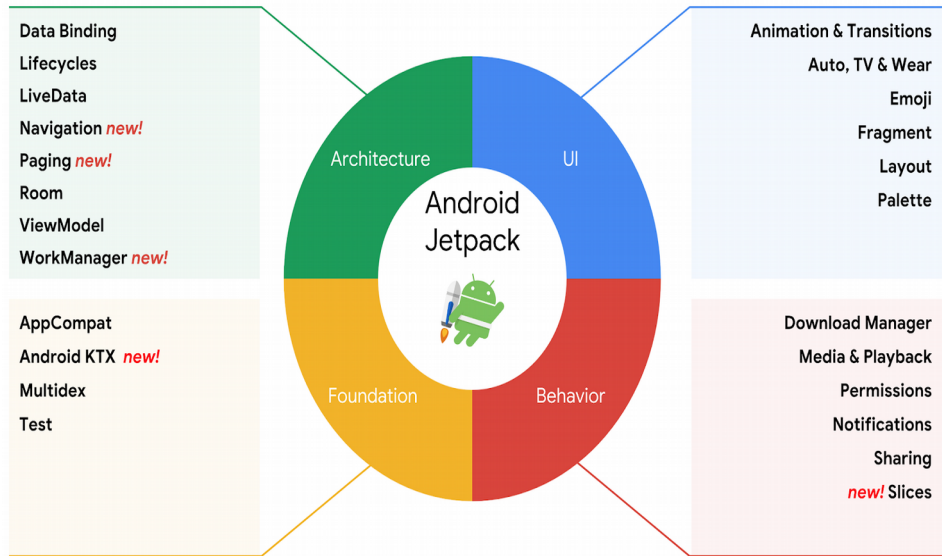




Android Jetpack

<https://developer.android.com/topic/libraries/architecture/>

Components



Android architecture components

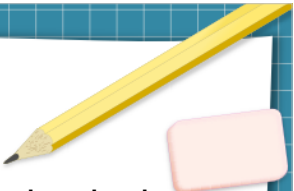


- Android architecture components are a collection of libraries that help you design robust, testable, and maintainable apps.
- Start with classes for managing your UI component lifecycle and handling data persistence.
 - Manage your app's lifecycle with ease. New lifecycle-aware components help you manage your activity and fragment lifecycles. Survive configuration changes, avoid memory leaks and easily load data into your UI.
 - Use LiveData to build data objects that notify views when the underlying database changes.
 - ViewModel Stores UI-related data that isn't destroyed on app rotations.
 - Room is an SQLite object mapping library. Use it to Avoid boilerplate code and easily convert SQLite table data to Java objects. Room provides compile time checks of SQLite statements and can return RxJava, Flowable and LiveData observables.

Introduction to Activities



- The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.
- Unlike programming paradigms in which apps are launched with a `main()` method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

- 
- The mobile-app experience differs from its desktop counterpart in that a user's interaction with the app doesn't always begin in the same place.
 - Instead, the user journey often begins non-deterministically.
 - For instance, if you open an email app from your home screen, you might see a list of emails.
 - By contrast, if you are using a social media app that then launches your email app, you might go directly to the email app's screen for composing an email.

Cont ...

- An activity provides the window in which the app draws its UI.
- This window typically fills the screen, but may be smaller than the screen and float on top of other windows.
- Generally, one activity implements one screen in an app.
- Each activity can then start another activity in order to perform different actions.

Cont ...

- Although activities work together to form a cohesive user experience in an app, each activity is only loosely bound to the other activities; there are usually minimal dependencies among the activities in an app.

Declare activities



- To declare your activity, open your manifest file and add an `<activity>` element as a child of the `<application>` element. For example:

```
<manifest ... >
```

```
<application ... >
```

```
    <activity android:name=".ExampleActivity" />
```

```
</application ... >
```

```
</manifest >
```

- The only required attribute for this element is `android:name`, which specifies the class name of the activity.

Declare intent filters

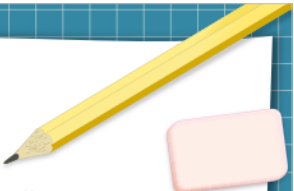


- Intent filters are a very powerful feature of the Android platform.
- They provide the ability to launch an activity based not only on an explicit request, but also an implicit one.
- For example, an explicit request might tell the system to "Start the Send Email activity in the Gmail app".
- By contrast, an implicit request tells the system to "Start a Send Email screen in any activity that can do the job."
- When the system UI asks a user which app to use in performing a task, that's an intent filter at work.

Cont ...

- You can take advantage of this feature by declaring an `<intent-filter>` attribute in the `<activity>` element.
- The definition of this element includes an `<action>` element and, optionally, a `<category>` element and/or a `<data>` element.
- These elements combine to specify the type of intent to which your activity can respond.
- For example, the following code snippet shows how to configure an activity that sends text data, and receives requests from other activities to do so:

Cont ...



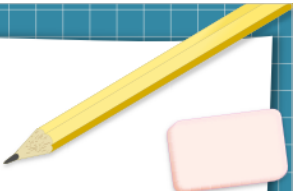
```
<activity android:name=".ExampleActivity"
android:icon="@drawable/app_icon">
<intent-filter>
    <action
android:name="android.intent.action.SEND" />
    <category
android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
</intent-filter>
</activity>
```

Cont ...



- In this example, the `<action>` element specifies that this activity sends data.
- Declaring the `<category>` element as `DEFAULT` enables the activity to receive launch requests.
- The `<data>` element specifies the type of data that this activity can send.

Cont ...



```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.setType("text/plain");
sendIntent.putExtra(Intent.EXTRA_TEXT,
    textMessage);
// Start the activity
startActivity(sendIntent);
```

Declare permissions



- You can use the manifest's <activity> tag to control which apps can start a particular activity.
- A parent activity cannot launch a child activity unless both activities have the same permissions in their manifest.
- If you declare a <uses-permission> element for a particular activity, the calling activity must have a matching <uses-permission> element.

Cont ...

- For example, if your app wants to use a hypothetical app named SocialApp to share a post on social media, SocialApp itself must define the permission that an app calling it must have:

```
<manifest>
```

```
<activity android:name="...."  
  android:permission="com.google.socialapp.per  
  mission.SHARE_POST" />
```

Cont ...

- Then, to be allowed to call SocialApp, your app must match the permission set in SocialApp's manifest:

```
<manifest>
```

```
  <uses-permission  
    android:name="com.google.socialapp.permission.SHARE_POST" />
```

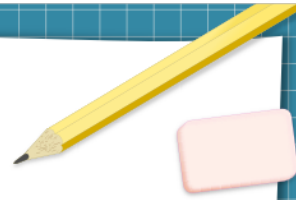
```
</manifest>
```

Managing the activity lifecycle



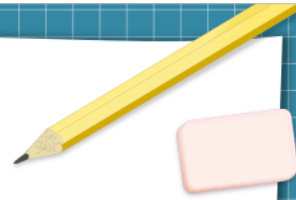
- Over the course of its lifetime, an activity goes through a number of states. You use a series of callbacks to handle transitions between states.
 - onCreate()
 - You must implement this callback, which fires when the system creates your activity.
 - Your implementation should initialize the essential components of your activity: For example, your app should create views and bind data to lists here.
 - Most importantly, this is where you must call setContentView() to define the layout for the activity's user interface.
 - When onCreate() finishes, the next callback is always onStart().

Cont ...



- **OnStart()**
 - As `onCreate()` exits, the activity enters the Started state, and the activity becomes visible to the user.
 - This callback contains what amounts to the activity's final preparations for coming to the foreground and becoming interactive.
- **OnResume()**
 - The system invokes this callback just before the activity starts interacting with the user.
 - At this point, the activity is at the top of the activity stack, and captures all user input.
 - Most of an app's core functionality is implemented in the `onResume()` method.
 - The `onPause()` callback always follows `onResume()`.

Cont ...



– onPause()

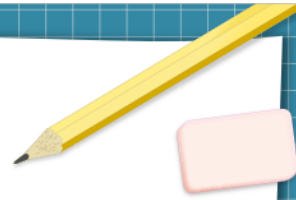
- The system calls onPause() when the activity loses focus and enters a Paused state.
- This state occurs when, for example, the user taps the Back or Recents button.
- When the system calls onPause() for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.
- An activity in the Paused state may continue to update the UI if the user is expecting the UI to update.
- **You should not use onPause() to save application or user data, make network calls, or execute database transactions.**
- Once onPause() finishes executing, the next callback is either onStop() or onResume(), depending on what happens after the activity enters the Paused state.

Cont ...

- onStop()

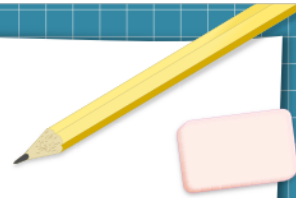
- The system calls `onStop()` when the activity is no longer visible to the user.
- This may happen because the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity.
- In all of these cases, the stopped activity is no longer visible at all.
- The next callback that the system calls is either `onRestart()`, if the activity is coming back to interact with the user, or by `onDestroy()` if this activity is completely terminating.

Cont ...



- OnRestart()
 - The system invokes this callback when an activity in the Stopped state is about to restart. onRestart() restores the state of the activity from the time that it was stopped.
 - This callback is always followed by onStart().
- OnDestroy()
 - The system invokes this callback before an activity is destroyed.
 - This callback is the final one that the activity receives. onDestroy() is usually implemented to ensure that all of an activity's resources are released when the activity, or the process containing it, is destroyed.

Cont ...



- Layouts (
<https://developer.android.com/guide/topics/ui/declaring-layout>
)
 - A layout defines the structure for a user interface in your app, such as in an activity.
 - All elements in the layout are built using a hierarchy of View and ViewGroup objects.
 - A View usually draws something the user can see and interact with.
 - Whereas a ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup objects

Cont ...

- The View objects are usually called "widgets" and can be one of many subclasses, such as Button or TextView.
- The ViewGroup objects are usually called "layouts" can be one of many types that provide a different layout structure, such as LinearLayout or ConstraintLayout
- You can declare a layout in two ways:
 - Declare UI elements in XML.
 - Instantiate layout elements at runtime

Load the XML Resource



- When you compile your app, each XML layout file is compiled into a View resource.
- You should load the layout resource from your app code, in your Activity.onCreate() callback implementation.
- Do so by calling setContentView(), passing it the reference to your layout resource in the form of: `R.layout.layout_file_name`

Attributes

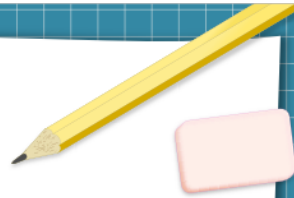


- Every View and ViewGroup object supports their own variety of XML attributes.
- Some attributes are specific to a View object (for example, TextView supports the textSize attribute), but these attributes are also inherited by any View objects that may extend this class.
- Some are common to all View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

Cont ...

- And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

ID



- Any View object may have an integer ID associated with it, to uniquely identify the View within the tree.
- When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the id attribute.
- This is an XML attribute common to all View objects (defined by the View class) and you will use it very often.
- The syntax for an ID, inside an XML tag is:
 - `android:id="@+id/my_button"`

Cont ...

- The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource.
- The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the R.java file).
- There are a number of other ID resources that are offered by the Android framework.
- When referencing an Android resource ID, you do not need the plus-symbol, but must add the android package namespace, like so:
 - `android:id="@android:id/empty"`

Layout Parameters

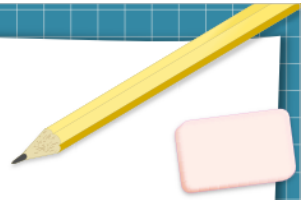


- XML layout attributes named `layout_something` define layout parameters for the View that are appropriate for the ViewGroup in which it resides.
- Every ViewGroup class implements a nested class that extends `ViewGroup.LayoutParams`.
- This subclass contains property types that define the size and position for each child view, as appropriate for the view group.

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.

Common Layouts



Linear Layout

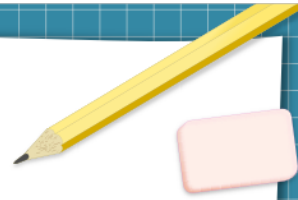
- A layout that organizes its children into a single horizontal or vertical row.
- It creates a scrollbar if the length of the window exceeds the length of the screen.

Relative Layout

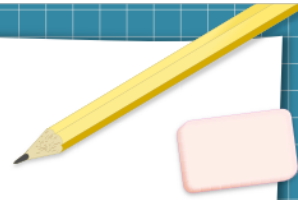
- Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).
-

Web View

- Displays web pages.

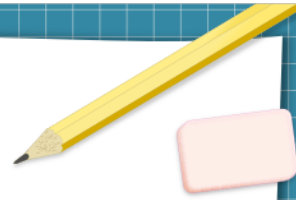


ConstraintLayout



- Is a ViewGroup which allows you to position and size widgets in a flexible way.
- Note: ConstraintLayout is available as a support library that you can use on Android systems starting with API level 9 (Gingerbread).
- There are currently various types of constraints that you can use:

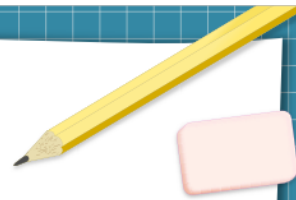
Cont ...



- Relative positioning
 - Allow you to position a given widget relative to another one.
 - You can constrain a widget on the horizontal and vertical axis:
- Margins
 - If side margins are set, they will be applied to the corresponding constraints, enforcing the margin as a space between the target and the source side.
- Centering positioning
- Circular positioning
 - You can constrain a widget center relative to another widget center, at an angle and a distance. This allows you to position a widget on a circle

Cont ...

- Visibility behavior
- Dimension constraints
- Chains
- Virtual Helpers objects
- Optimizer



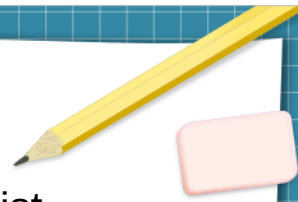
Layouts with an Adapter



- When the content for your layout is dynamic or not pre-determined, you can use 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.

List View

- Displays a scrolling single column list.



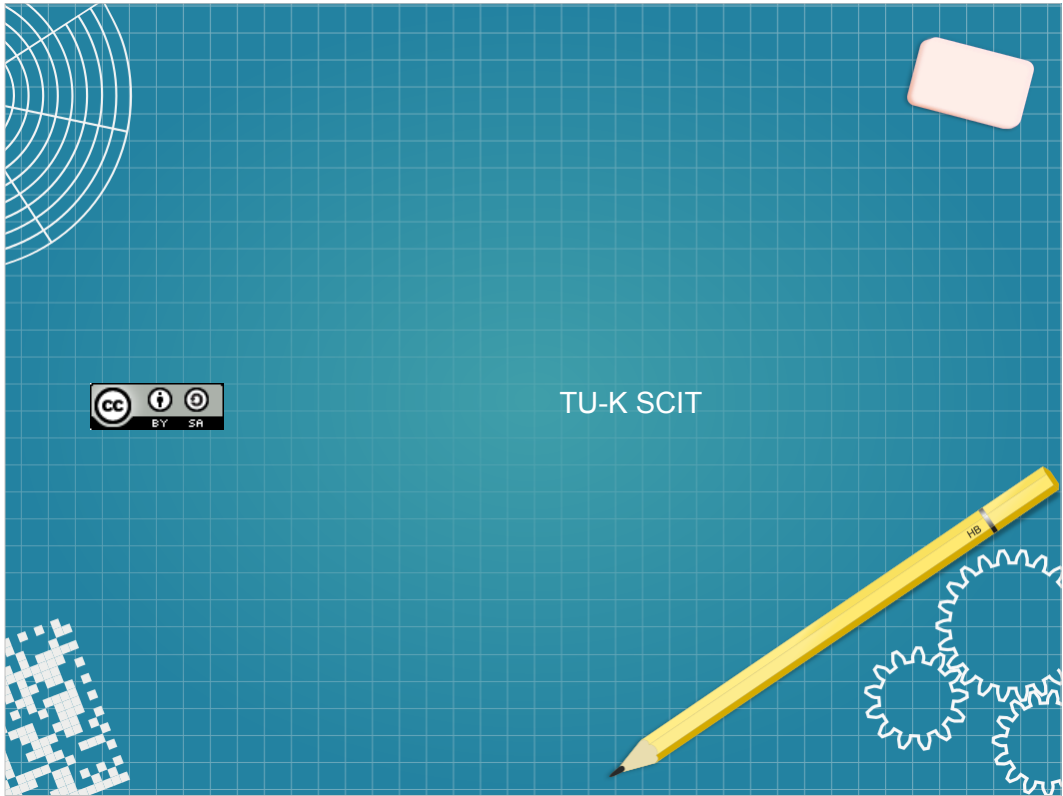
Grid View

- Displays a scrolling grid of columns and rows

Filling an adapter view with data



- You can populate an AdapterView such as ListView or GridView by binding the AdapterView instance to an Adapter, which retrieves data from an external source and creates a View that represents each data entry.
- Android provides several subclasses of Adapter that are useful for retrieving different kinds of data and building views for an AdapterView. The two most common adapters are:
 - ArrayAdapter
 - SimpleCursorAdapter



TU-K SCIT