Module 6

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 6 Study Guide and Deliverables

Topics:	• Notifications and Alarms • Android Application Security • Intro to Graphics
Readings:	• Online lectures • Notes and references cited in lectures
Discussions:	• Discussion 6 postings due **Tuesday, August 16 at 6:00 ET**
Labs:	• No labs due this module.
Assignments:	• Assignment 6 due **Tuesday, August 16 at 6:00 am ET** • Project Presentation Video **Sunday, August 14 at 6:00am ET**
Assessments:	• Quiz 6 due **Tuesday, August 16 at 6:00am ET**
Course Evaluation:	Course Evaluation opens on **Tuesday, August 9 at 10:00 AM ET** and closes on **Tuesday, August 16 at 11:59 PM ET**. Please complete the course evaluation. Your feedback is important to MET, as it helps us make improvements to the program and the course for future students.
Live Classrooms:	• **Tuesday, August 15 from 6:00-8:00pm ET**

Learning Outcomes

By the end of this module, you will be able to:

- Compare normal intents and pending intents, and describe how to use pending intents in notifications and alarms.
- Describe what is a notification and how to use notification to create a foreground service.
- Describe various Android security issues, such as storage security, permissions and open components.
- Describe what is a Drawable and how to use it to draw static graphics or animations in a view.
- Describe what is a Canvas and how to use it to draw dynamic graphics in a customized view.

Introduction

In this module, we will discuss several topics. First we will show how to use notifications and alarms. Then we will discuss some tips to avoid security vulnerabilities in Android apps. Lastly we will cover basic graphics and animations.

Topic 1: Notifications and Alarms

PendingIntent

Before we introduce notifications and alarms, we need to discuss pendingIntents, for both of them using pendingIntents to specify their behaviors.

A PendingIntent object is a wrapper around an Intent object. When using a pendingintent, your app doesn't send the intent immediately with a call such as `startActivity()`. Instead, **it specifies its included intent and hands it to another foreign application (e.g. NotificationManager, AlarmManager, Home Screen AppWidgetManager, or other 3rd party applications) who will send the intent on behalf of your app.** A pending intent is a token given to a foreign application (e.g. NotificationManager) to grant it the same permissions and identity, so that it can perform some operations on behalf of your app. Therefore, pendingIntents should be used with extra caution. Explicit intents should be wrapped in the pendingIntent. Implicit intents should be avoided.

PendingIntent is a final class defined in Android. To create a pendingIntent object, we need to call one of the following static methods from the PendingIntent class depending on whether the contained intent should be sent to an activity, a service, or broadcast receivers later.

- PendingIntent getActivity(Context!, Int, Intent!, Int): Retrieve a PendingIntent that will start a new activity, like calling Context.startActivity(Intent).
- PendingIntent getActivity(Context!, Int, Array<Intent!>, Int): Retrieve a PendingIntent that will start a number of activities, like calling Context.startActivities(Intent[]).
- PendingIntent getBroadcast (Context!, Int requestCode, Intent!, Int): Retrieve a PendingIntent that will perform a broadcast, like calling Context.sendBroadcast(Intent).
- PendingIntent getService (Context!, Int requestCode, Intent, Int): Retrieve a PendingIntentthat will start a service , like calling Context.startService(Intent).

Here are common uses of pendingIntents:

- Declaring an intent to be executed when the user performs an action with your Notification (the Android system's NotificationManager executes the contained Intent).
- Declaring an intent to be executed when the user performs an action with your App Widget (the Home screen app executes the contained Intent).
- Declaring an intent to be executed at a specified future time (the Android system's AlarmManager executes the contained Intent).

Test Yourself

What is the relationship between a normal intent and a pending intent?

A pending intent wraps a normal intent, and is delivered to a foreign app who will send the contained intent on behalf of your app.

Test Yourself

Why can it be dangerous using a pending intent?

Because the foreign application is granted the same permissions and identity as your app. Potentially it can do harmful things on behalf of your apps and access information of your app.

Notifications

Notifications are used to display messages outside the application's UI. It can notify the user of timely events in the background. Users can tap the notification to open the app or take an action directly from the notification. They can be managed far more than Toast. Notifications usually take the form of a persistent icon in the status bar or an entry in the notification drawer, possibly with flashing backlight, sound, and vibrations. Both the notification area and the notification drawer are system-controlled areas that the user can view at any time. From the last module, we know notifications are required in foreground services.

Notification is a class defined in the android.app package to represent a persistent notification. It has several nested classes, such as Notification.Builder and Notification.Action.

`Notification.Builder` is used to help construct a notification. The builder pattern is used here to help the complex construction process. It provides a number of methods to set the attributes and add actions to the notification to be built. Starting in Android 8.0 (API level 26), all notifications must be assigned to a channel. To create a Notification.Builder (or NotificationCompat.Builder) object, in Android 8.0 or above, we need to pass two parameters into the constructor method, while for lower API levels, only the context is needed. We can use `NotificationCompat.Builder` from androidx library for the backwards compatibility.

We can then set the content of the notification and other attributes using the builder object. For example, we can dynamically check the sdk version of the device and decide whether we need a notification channel.

```
fun buildNotification(): Notification {

   val channelId = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
       val newChannelId = "ChannelId"
       val channelName = "My Background Service"
       val channel =
           NotificationChannel(newChannelId, channelName,
               NotificationManager.IMPORTANCE_DEFAULT)
       val service = getSystemService(Context.NOTIFICATION_SERVICE) as
               NotificationManager
       service.createNotificationChannel(channel)
       newChannelId
   } else {
       ""
   }

   val notificationIntent = Intent(this,
       MainActivity::class.java)

   val pendingIntent: PendingIntent = PendingIntent.getActivity(
       this, 0,
       notificationIntent, 0
   )
   return NotificationCompat.Builder(this, channelId)
```

```
        .setContentTitle("Example Notification")
        .setTicker("Example Notification")
        .setContentText("My Notification")
        .setSmallIcon(R.drawable.ic_service)
        .setContentIntent(pendingIntent)
        .setOngoing(true)
        .build()
}
```

In the above code snippet, we called several setter methods to specify how to build the notification. Each setter method returns the Notification.Builder object again.

Not only displaying a message, every notification should respond to a tap. This is done through defining a pendingIntent object and passing it to `setContentIntent()`. So when the user clicks on the notification, the notification manager will send the contained intent on behalf of our app. We can also add more actions to the notification, by providing an action icon and a pendingIntent for each action.

To issue a notification, we need to call notify(int, Notification!) of the NotificationManager object.

```
notificationManager = (NotificationManager)context.getSystemService(NOTIFICATION_SERVICE);
notificationManager.notify(0, mNotification);
```

Test Yourself

The _____ class is used to build a notification. Its _____ method is called to return a notification object. The _____class is used to issue a notification. Its ____ method is called.

Notification.Builder, build(), NotificationManager, notify()

Alarms

Android system provides alarm services which schedule events in a future time .They are orchestrated via the `AlarmManager`, which allows an Intent to be sent at a designated time.

`AlarmManager` is a class also defined in the android.app package, providing access to the system alarm service. To set an alarm, we need first obtain the reference to the AlarmManager by calling context.getSystemService(Context.ALARM_SERVICE), then we need to specify the alarm time and provide a pendingIntent. When an alarm is fired up, the intent wrapped in the pendingIntent is sent by the system, automatically starting the target component even if it is not already running.

The following code snippet shows an example of using Android alarm service to schedule an operation in the monitor service. We can set an alarm after a particular delay or at a specific time. We can also set a repeated alarm.

```
val mAlarmManager = getSystemService(ALARM_SERVICE) as AlarmManager
```

```
    //create a pendingIntent to specify the action for the alarm
    val serviceIntent = Intent(this, MyFgService::class.java)
    serviceIntent.action = ACTION.PLAY_ACTION
    val pendingIntent = PendingIntent.getService(
        this, 0,
        serviceIntent, 0
    )

    // cancel the previous alarms
    mAlarmManager.cancel(pendingIntent)

    // set the alarm after 5 second
    mAlarmManager[AlarmManager.RTC_WAKEUP, 5000] = pendingIntent
    // set a repeated alarm after 5 second with every 15 minutes
    //mAlarmManager.setRepeating(AlarmManager.RTC_WAKEUP, 5000,
    //    AlarmManager.INTERVAL_FIFTEEN_MINUTES, pendingIntent)

}
```

In the last module, we learned that we can also use a handler to schedule an event in the future. Compared with handlers, using AlarmManager results in much higher overhead. Therefore, the alarm should be set at minute level. If you want to have a periodic alarm. The period should be at least 15 minutes. However, the handler does not work well if the application is not running, or the device is in sleep mode. The handler uses uptime while AlarmManager can use real time. Please check the developer website to find more details on when to use AlarmManager at Schedule alarms.

Test Yourself

Complete the following code:

val mAlarmManager = (_____) _____ ;

val mNotificationManager = (_____) _____ ;

AlarmManager, getSystemService(Context.ALARM_SERVICE, NotificationManager, getSystemService(Context.NOTIFICATION_SERVICE)

Test Yourself

What does the following code snippet do?

```
val intent = Intent(this, ForegroundService.class)
intent.setAction("actoin1")
val pendingIntent = PendingIntent.getService(this, 1, intent, 0 )

getSystemService(Context.ALARM_SERVICE).set(AlarmManager.ELAPSED_REALTIME_
       SystemClock.elapsedRealtime() + AlarmManager.INTERVAL_FIFTEEN_MINUT
```

Start foreground service with action1 after 15 minutes.

Topic 2: Android Application Security

Android Application Security

A high quality application not only implements the functionalities correctly for users, but also has good performance and security. In this module, we will discuss some security tips that help avoid security vulnerabilities.

Android is designed with security in mind. It uses a number of security mechanisms to protect user data, systems and applications. The detailed information about Android security mechanisms can be found on the Android developer website. The Android developer website also provides several security tips in application development.In this module, we will introduce three mechanisms provided by Android

Storage Isolation

Android provides an OS-level sandbox for each application. The application sandbox provides application isolation, preventing apps from accessing each other's memory and files. This sandbox is based on basic process isolation and user-based data/file protection mechanisms supported by Linux. Since by default each application runs in a separate process, they cannot access each other's memory. Android also leverages basic Linux file system protection, but each application is assigned to a different uid and gid. Its internal storage space under **/data/data/<appfullpackagename>/** is only accessible to its owner, the app uid. Therefore, it cannot be accessed by other applications. As shown in the following screenshot, each application folder's uid is different and each folder is only accessible by the owner.

```
drwx------  4 u0_a96 u0_a96 4096 2018-04-28 17:12 edu.bu.myapplication1
drwx------  6 u0_a92 u0_a92 4096 2018-04-25 13:28 edu.bu.projectportal
drwx------  4 u0_a89 u0_a89 4096 2018-04-22 04:50 edu.bu.testmfcc
drwx------  4 u0_a95 u0_a95 4096 2018-04-28 17:05 edu.bu.threadhandlerexample
drwx------  4 u0_a94 u0_a94 4096 2018-04-26 04:50 edu.bu.useprojectportalprovider
drwx------  4 u0_a86 u0_a86 4096 2018-04-03 19:39 edu.bu.widgetsexplore
```

However, the external storage space in /sdcard/ is accessible to everyone. Therefore, when storing data locally, we should use the internal storage. Only use the external storage when you really need to make your data universally accessible. Never store any sensitive information using external storage. **If you need to read data from external storage, always perform input validation or verification as data can be from untrusted sources.** In addition, when using the internal storage, avoid MODE_WORLD_WRITEABLE or MODE_WORLD_READABLE modes for IPC files. These constants are actually deprecated in API 17. If you want to share your data with other apps, we should use a content provider and also clearly specify the read and write permissions, in order to grant the access to others only as needed.

Test Yourself

How does Android provide file/data isolation for different applications?

By assigning each app a different uid and it folder is only accessible by its uid.

Test Yourself

Data that is only accessed by its own app should be stored using _____. Data that needs to be accessed by some other application should be stored using _____. Data that can be accessed by any application are stored using _____ .

internal storage, content provider with proper permission, external storage

Use Permissions

Android Permission Model

Android permission Model Android uses a permission model to protect system resources as well as the communication between applications. Probably this is the most discussed topic by the Android security research community in the past several years. Many have criticized the permission system used in versions prior to Android 6, and many enhancements have been proposed from the research community. Finally, Android 6 starts to support the runtime partial permission model.

The idea of using permissions is based on the least privilege principle. By default, an application can only access a limited range of system resources. Additional permissions are required to access more privileged resources. This can help prevent apps from incorrectly or maliciously accessing many resources. These resources can be categorized into four groups:

- Cost-sensitive services such as telephony, SMS/MMS, Network/Data, In-App Billing or NFC Access.
- Personal Information such as the phone book or the calendar.
- Sensitive Data Input devices such as GPS, camera, or microphone.
- Device Metadata such as system logs, browser history, phone numbers, or hardware / network identification information.

To access these resources, the application needs to declare the corresponding permissions explicitly in the manifest file. Prior to Android 6, the user is prompted with a screen with all requested permissions at the installation time and asked whether to continue the installation. The user can either cancel the installation, or grant all permissions to continue the installation. The user can not grant or deny individual permissions prior to Android 6. Once granted, the permissions are applied to the application as long as it is installed. Google argues that this model provides better user experience, enabling seamless switch between applications at will. However, this “all or nothing” installation permission mechanism can be exploited by various attacks, and has been criticized by many researchers.

Finally Android 6 enables a runtime permission model, where the application requests the permissions at run time (the first time), and the user can accept or deny individual permissions and also turn on and off individual permission for all installed apps later. For apps designed for releases prior to Android 6, when used with Android 6 or above:

- The application requests all permissions at the installation time.
- The application is granted with all permissions if the installation continues.
- The user can turn on and off individual permissions through device setting after the installation.

For apps designed for Android 6 or above, being used with Android 6 or above:

- **The app only requests permissions when it needs them at run time.**
- The user can deny a permission and still continue to use the app.
- The user can turn permissions on and off for installed apps later.

Request permissions at Runtime

All permissions need to be declared in the manifest file. For example, if we need to use the internet and make phone calls, we need to declare the following permissions in the manifest file.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.CALL_PHONE" />
```

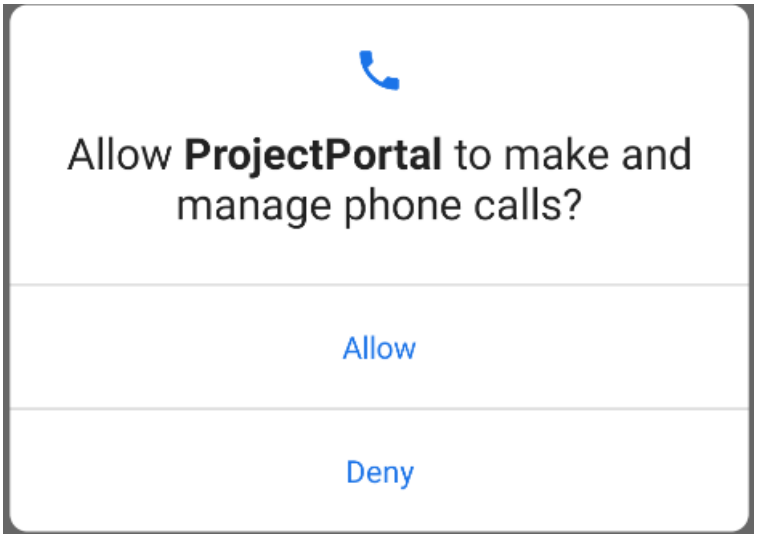
If it is for Android 5 or lower, that is all we need to do regarding permissions. However, if we use Android 6 or above, we need to also **request dangerous permissions at the runtime.** Android differentiates normal permissions from dangerous permissions. Normal permissions such as the INTERNET permission are granted automatically without needing the approval from the user. However, dangerous permissions such as the CALL_PHONE permission need explicit approval by the user at runtime. When calling these APIs that need dangerous permissions, we should always check and request the permissions at runtime. Whenever a permission is required, the android framework layer will check if the application is granted with that permission. If yes, the execution continues. If not, a security Exception is thrown back to the application. If the application does not handle it correctly, it may cause the app to crash or terminate.

The following code shows how to check and request the CALL_PHONE permission at run time. The runtime permission request is only needed on the device with Android M or above. To check the permission, we need to call `ContextCompat.checkSelfPermission()`. To request a permission we can call requestPermissions() by ourselves, or use the RequestPermission contract. The following code snippet shows how to use the RequestPermission contract which is recommended due to its simplicity. First we need to create a RequestPermission contract using its default constructor, then we need to create a RequestPermissionLauncher which is returned from registerForActivityResult() when we register a ActivityResultCallback for it. The callback method takes one boolean parameter as the input which indicates if the user allows or denies the request, and we shall define how we handle the user's response to the permission request. At the runtime, when we need to request the permission, we can simply call the launch() method of the requestPermissionLauncher we created before. For simplicity, we simply display a toast message when the user denies the permission request, or when the app tries to show the rationale of the permission in the following code. Better UI is recommended to provide users better understanding of the permission used, such as an Alert dialog. For more details about how to request permissions, please read Request app permissions. One thing we should be aware of is that Starting in Android 11 (API level 30), if the user taps Deny for a specific permission more than once during your app's lifetime of installation on a device, the user doesn't see the system permissions dialog if your app requests that permission again.

```
val requestPermissionLauncher =
   registerForActivityResult(ActivityResultContracts.RequestPermission()) {
           isGranted ->
       if (isGranted) {
           startActivity(Intent(Intent.ACTION_CALL,
               Uri.parse("tel:" + phoneNum)))
       } else {
           Toast.makeText(this,
               "we cannot make phonecall without permission granted",
               Toast.LENGTH_LONG).show()
       }
   }


@RequiresApi(Build.VERSION_CODES.M)
fun callUs(v: View) {
   if (ContextCompat.checkSelfPermission(this,
           Manifest.permission.CALL_PHONE) ==
       PackageManager.PERMISSION_GRANTED)  {
       startActivity(Intent(Intent.ACTION_CALL,
           Uri.parse("tel:" + phoneNum)))
   } else {
       if (shouldShowRequestPermissionRationale(Manifest.permission.CALL_PHONE)) {
```

```
            Toast.makeText(this, "we need permission to make phone call",
                Toast.LENGTH_LONG.show()
        }
        requestPermissionLauncher.launch(Manifest.permission.CALL_PHONE)
    }
}
```

When the requestPermissionLauncher is launched, a system dialog appears on the screen.



Besides system default permissions, an application may declare their own permissions for other applications. Permissions are applied not only for accessing protected system resources, but also application communication. Permissions are used under the following circumstances:

- When executing certain security sensitive functions (only apply to system default permissions).
- When starting an activity if it is specified with any permission(s).
- When starting or binding a service if it is specified with any permission(s).
- When sending or receiving broadcasts if it is specified with any permission(s).
- When accessing a content provider if it is specified with any permission(s).

For example, we have a broadcast receiver in our app that can send SMS messages. Instead of allowing any application to broadcast an intent to it (and thus send SMS on their behalf), the sender must also have the SEND_SMS permission using the <android:permission> tag as follows. This can prevent a malicious app without such permission from misusing our receiver to send SMS messages on its behalf.

```
<receiver
    android:name=".MyReceiver"
    android:enabled="true"
    android:permission="android.permission.SEND_SMS">
    <intent-filter>
        <action android:name="android.intent.action.BATTERY_LOW"></action>
        <action android:name="edu.bu.databaseexample.broadcastassignment"></action>
    </intent-filter>
</receiver>
```

Test Yourself

Only dangerous permission need be declared in the manifest file

True

False

All permissions need to be declared.

Test Yourself

When should we request a permission at runtime?

Device sdk version is android M or above, dangerous permissions, not granted before.

Secure Inter-Component Communication

Instead of using traditional Linux IPCs, such as network sockets and shared files, we should use Android-specific IPC APIs, such as Intent, Binder, or Messenger with a Service and BroadcastReceiver. Android IPC (or ICC) provides better control and security.

The Open Component Issue

In general, an Android component (an activity, a service, or a content provider) should be private and not be used by other apps. The `android:exported` attribute should be false in the manifest file. If this attribute is set to `true`, we call it an open component, and can be accessed by other applications through IPC. Activities and services are not exported by default. Broadcast receivers are usually exported so that it can receive broadcast intents. Content providers are not open components by default in Android 7 or above. But they are exported by default in the Android M or lower versions. Therefore, explicitly setting this attribute to be `false` is a good security practice. If a component is set with an intent filter, then it also becomes an open component so that it can receive that type of intent from others. In this case, it is better protected by some permission, as shown in the previous receiver example.

To improve the security, **in Android 12 or above, the default value for the exported attribute of all components is false. You need to explicitly set the `android:exported` to be true even if the intent filter is defined, in order to be accessible by other applications**.

Explicit Intents and Implicit Intents

The intent filter can unnecessarily expose the component to be accessible by others. Intents are used in Android for inter component communication. If the target receiver component is specified in the intent, Android system knows exactly who the intent should be delivered to. These are explicit intents. However, if the target receiver component is Not specified in the intent, Android system has to figure out based on all open components' intent filters. A malicious app can possibly hijack the intents, sending incorrect data to or receiving sensitive data from the sending component. Implicit intents should be avoided in general unless it is really necessary, particularly, in pending intents.

Other security tips includes:

- Always use parameterized queries on databases or content providers to avoid SQL injection attack.
- Apply additional crypto functions on sensitive information.

Test Yourself

If a component has an intent filter set, it is an ____ component and can receive intents from others in Android 11 or below.

Open

Test Yourself

It is a good practice to protect open components with _____.

Permissions

Test Yourself

An _____ intent can be hijacked by malicious apps.

Implicit

Topic 3: Intro to Graphics

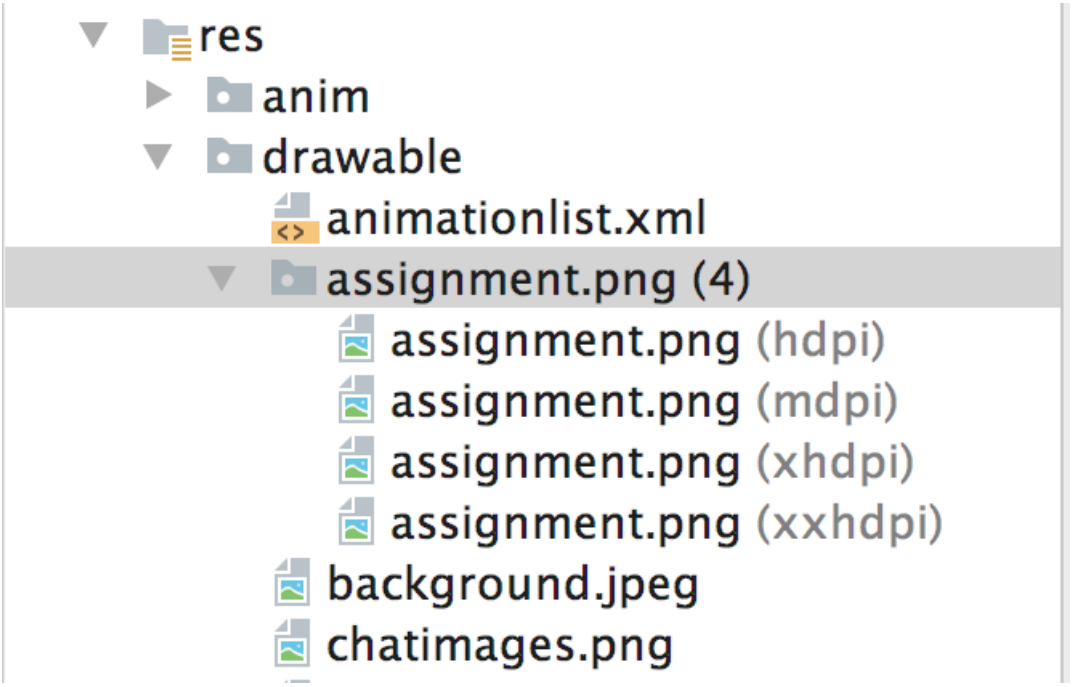
Intro to Graphics

The last topic to cover in this course is Android graphics, mainly focusing on static graphics or predefined animations. These are simple graphics displayed within a relatively static app that don't need to change dynamically. Mostly they are drawn into a View object from the layout and handled by the system's normal View hierarchy. Common packages used for 2D drawing and animating are `android.graphics.drawable` and `android.view.animation`. Another way to draw graphics is to draw directly onto a Canvas. We will also discuss it briefly.

Drawable

A drawable is a general abstraction for "something that can be drawn". It is only for display purposes. It cannot receive events or interact with the user. Drawable is an abstract class defined in the `android.graphics.drawable` package. It has several subclasses: BitmapDrawable, ShapeDrawable, PictureDrawable, NinePatchDrawable, LayerDrawable, TransitionDrawable, AnimationDrawable, etc.

The simplest drawable is a static graphical file (e.g. a bitmap image), which would be represented in Android via a BitmapDrawable class. Drawables can be stored as individual files in one of the `res/drawable` folders. Bitmap files (png, jpg) are usually stored for different resolutions in the -mdpi, -hdpi, -xhdpi, -xxhdpi subfolders of res/drawable. The Android system selects the correct one automatically based on the device configuration. For example, four assignment.png files are stored in the drawable folder with different resolutions in different subfolders.

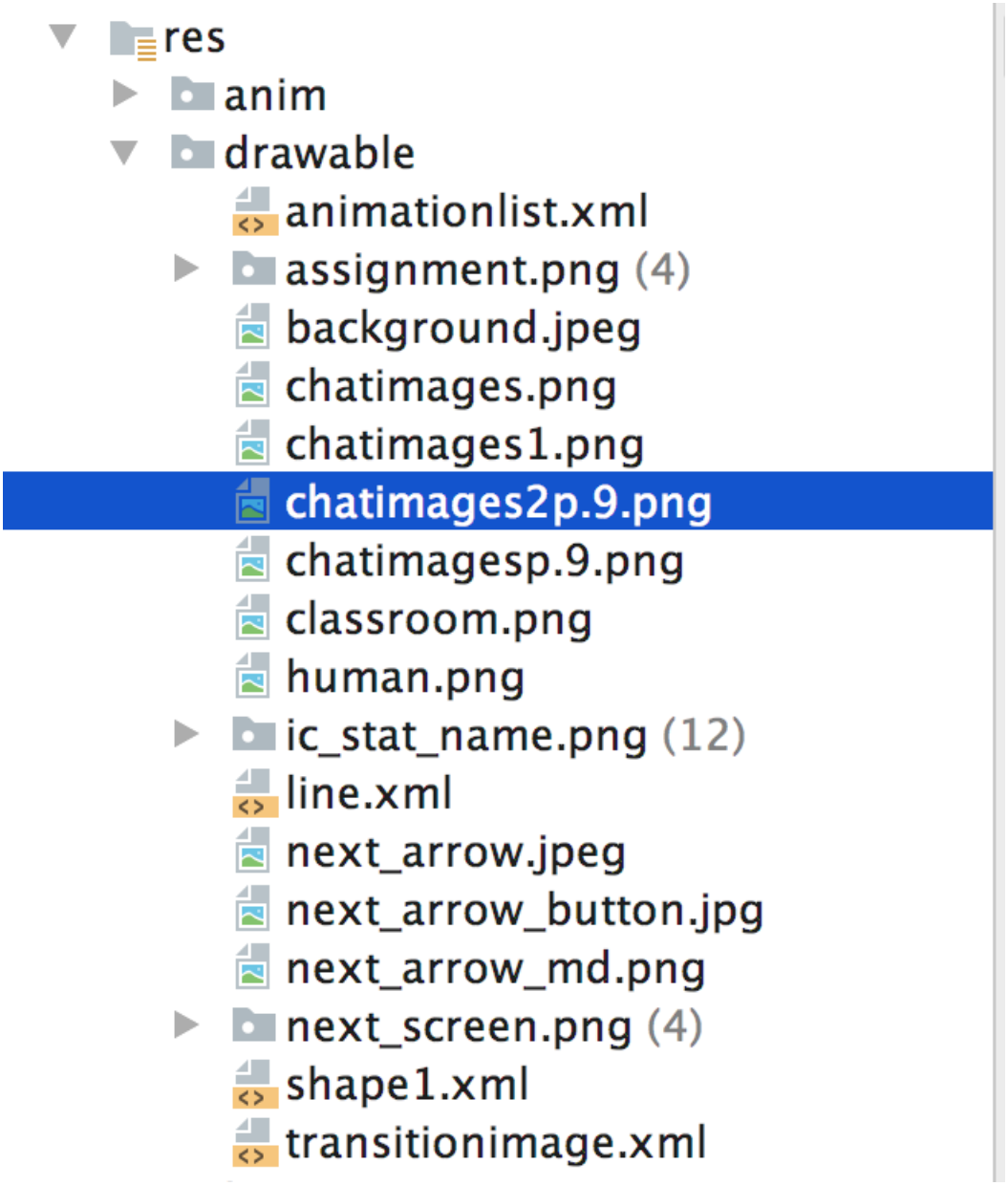


XML drawables are used to describe shapes in terms of color, border, gradient, state, transitions and more. For example line.xml and shape1.xml are ShapeDrawables defined in xml stored in this folder. The shape1.xml is shown as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android" type="oval" >
   <solid android:color="#FFFF0000"/>
   <padding android:left="10sp" android:top="4sp"
       android:right="10sp" android:bottom="4sp" />
   <stroke android:width="1dp" android:color="#00000000"/>
</shape>
```

It defines a custom shape, specifying its color, padding, type and stroke width. It basically creates a ShapeDrawable object when the xml file is loaded and inflated.

9-patch graphics are used to define which part of a graph should be stretched if the view which uses this graphic is larger than the graph. For example, chatimages2p.9.png is a 9-patch file. Android provides a built-in tool to help you create a 9-patch file. Right click on an image file, you can find the “Create 9-patch files … ” menu item in the pop-up menu.



Drawables can also be created directly in Java/kotlin code using predefined Drawable subclasses. We can also create custom Drawables by extending the Drawable Class. Drawable objects are then used by View objects.

Using Drawables in Views

A drawable stored in the res/drawable folder is referred via its resource id, which is the filename without the file extension. It can be referred to in the XML layout file via @drawable/resourceid. It can also be referred via R.drawable.resourceid in Java/kotlin code. For example, the resource id of the next_arrow.jpeg file is next_arrow. To use it as the source image file of an image button, we can directly refer to it via **@drawable/next_arrow**.

```
<ImageButton
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:src="@drawable/next_arrow"
   android:onClick="nextScreen"/>
```

In the XML layout, the resource id can be used in some attributes such as `android:background` or `android:src`. Imageviews can use a drawable as its image source. Most views can set its background to a drawable.

We can also refer to it in Java/kotlin code via R.drawable.next_arrow. For example,
`imageBtn.setImageResource(R.drawable.next_arrow)`

We can also create a drawable object and then pass it to views. For example,

```
ResourcesCompat.getDrawable(getResources(),
R.drawable.next_arrow, null);
imageBtn.setImageDrawable(imageDrawable);
```

The getResources() method can be called from a context to get the resource object stored in the res/ folder. Then getDrawable() can get a drawable object stored in the res/drawable/ subfolder based on its resource id. Any Drawable subclass that supports inflate() can be defined in the XML layout and instantiated.

Drawable objects are commonly used in image views (such as ImageView objects and ImageButton objects) which can load images from various sources such as resources or content providers to display. They can compute measurements from an image and provide display options such as scaling and tinting. Drawables are also commonly used as background of other views. For example, we can set a text view's background to be an image defined by shape1.xml in the previous section.

```
findViewById<TextView>(R.id.textViewid).setBackgroundResource(R.drawable.shape1)
```

While in most applications, we use static XML layout to define the look. We can always dynamically add a view element or customize views to draw dynamically.

For example, the following code snippet is called when a button "Add View" is clicked. It dynamically adds another image view into a linear layout.

```
fun addImageView(v: View?) {
   // Instantiate an ImageView and define its properties
   val iV = ImageView(this)
   iV.setImageResource(R.drawable.human)
   iV.adjustViewBounds = true // set the ImageView bounds to match Drawable 's dim's
   iV.layoutParams = LinearLayout.LayoutParams( // width and height
       LinearLayout.LayoutParams.WRAP_CONTENT, LinearLayout.LayoutParams.WRAP_CONTENT
   )
   // Add the ImageView to the layout and set the layout as the content view
   findViewById<LinearLayout>(R.id.linearlayout)?.addView(iV)
```

We can also make a custom view and override the constructor method and the onDraw() method to perform customized drawing. The custom view can then be used in XML or Java code directly. We can first create a shape drawable to draw on the canvas that is passed in through

onDraw() call back. We can also create a paint and use the various drawing methods of the canvas to draw an oval.

```
class CustomDrawableView(context: Context?) :
   View(context) {
   override fun onDraw(canvas: Canvas) {
       createDrawable().draw(canvas)
      // drawWithPaint(canvas)
   }

   fun createDrawable(): ShapeDrawable {
       // draw a oval
       val mDrawable = ShapeDrawable(OvalShape())
       //the shape cannot be drawn without the bound set
       mDrawable.setBounds(10, 10, 610, 110)
       // the default color is black if not set
       mDrawable.paint.color = Color.GREEN
       return mDrawable
   }

   fun drawWithPaint(canvas: Canvas){
       // we can also use paint directly draw on canvas
       val myPaint = Paint(Paint.ANTI_ALIAS_FLAG);
       myPaint.color = Color.GREEN
       myPaint.style = Paint.Style.FILL
       canvas.drawOval(RectF(10.0f, 10.0f, 610.0f, 110.0f),
           myPaint);

   }
}
```

Test Yourself

If we need to use a file background.jpg as the background of the screen, we can store the file in _____. It can be referred to in XML via _____, and java/kotlin code via _____. We can use the ____ attribute in XML to set the background of the _____.

res/drawable, @drawable/background, R.drawable.background, android:background, root level layout

Canvas

In the above example, the onDraw() method takes a Canvas object as its parameter. When an app requires specialized drawing and/or control the animation of graphics, we need to use canvas upon which the customized graphics are drawn. Canvas is a class defined in the

android.graphics package for custom drawing. It basically holds different “draw” calls.

Android provides each View object with a canva that the onDraw() method uses. The above custom View class implements the onDraw(Canvas) callback which draws **on the default canvas** of the View. The onDraw(Canvas) is called by the Android system only when necessary. To redraw your UI screen, you must first call invalidate(). Then Android calls your view's onDraw(Canvas) method, though this call isn't guaranteed to be instantaneous.

Drawable has its own draw() method with a Canvas argument. To use the ShapeDrawable object’s draw() method, we can pass the View’s canvas to it, as shown in the above example. We can also use various drawing methods of the canvas itself.

Canvas provides a lot of drawing methods that you can use to draw different pictures. For example, drawArc(), drawBitmap(), drawColor(), drawLines(), drawOval(), drawPath(), drawPicture(), drawRect(), etc. The following class defines a custom view used to implement a simple drawing app. It is adapted from Android Canvas Example

```
class CustomCanvasView(context: Context?) : View(context) {
    private val mPath: Path
    private val mPaint: Paint
    private var mX = 0f
    private var mY = 0f
    override fun onDraw(canvas: Canvas) {
        super.onDraw(canvas)
        canvas.drawPath(mPath, mPaint)
        mPaint.style = Paint.Style.STROKE
    }

    private fun startTouch(x: Float, y: Float) {
        mPath.moveTo(x, y)
        mX = x
        mY = y
    }

    private fun upTouch() {
        mPath.lineTo(mX, mY)
    }

    private fun moveTouch(x: Float, y: Float) {
        val dx = Math.abs(x - mX)
        val dy = Math.abs(x - mY)
        if (dx >= 5 || dy >= 5) {
            mPath.quadTo(mX, mY, (x + mX) / 2, (y + mY) / 2)
            mX = x
            mY = y
        }
    }

    fun clearCanvas() {
```

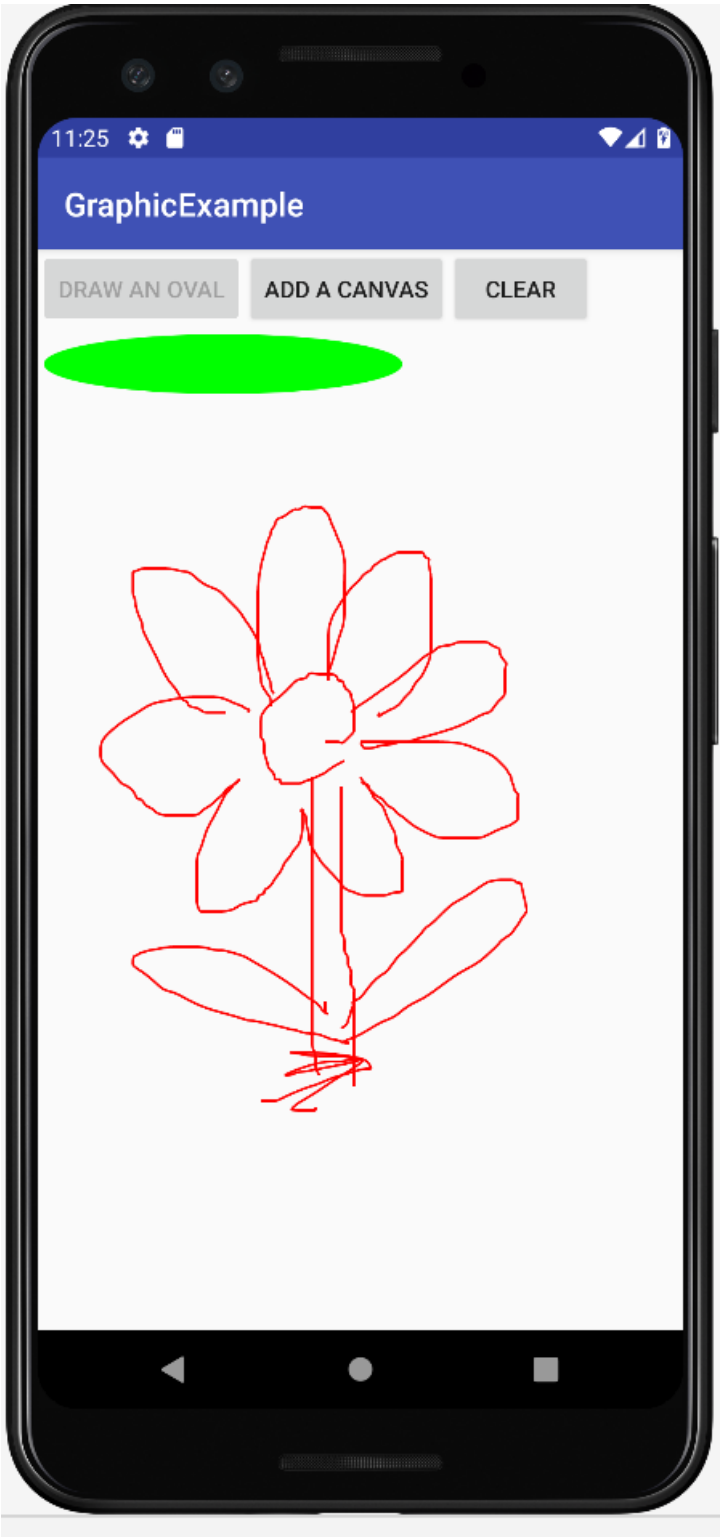
```
        mPath.reset()
        invalidate()
    }

    fun myOnTouchEvent(event: MotionEvent): Boolean {
        val x = event.x
        val y = event.y
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                startTouch(x, y)
                invalidate()
                Log.d("Canvas", "down $x $y")
            }
            MotionEvent.ACTION_MOVE -> {
                moveTouch(x, y)
                invalidate()
                Log.d("Canvas", "move $x $y")
            }
            MotionEvent.ACTION_UP -> {
                upTouch()
                invalidate()
                Log.d("Canvas", "up $x $y")
            }
        }
        return true
    }

    init {
        mPath = Path()
        mPaint = Paint(Paint.ANTI_ALIAS_FLAG)
        mPaint.color = Color.BLACK
        mPaint.strokeWidth = 4f
    }
```

The myOnTouchEvent() method is called in the onTouch() callback defined by CustomCanvasView.OnTouchListener. It responds when the user touches on screen.

```
myCanvas!!.setOnTouchListener {
        v, e -> (v as CustomCanvasView).myOnTouchEvent(e) }
```

It uses the drawPath() method of the default canvas. When invalidate() is called, the system will call the custom onDraw() method. It basically traces all user touch pointer movements. Here is the “not so pretty” flower I drew on the screen using this app. The clearCanvas() method is

called when the clear button is clicked to clear the canvas.



Test Yourself

To perform a custom drawing, we need to use _____. A view has a default _____ to draw on. To redraw a view, we need call _____ , which in turn calls _____.

canvas, canvas, invalidate(), onDraw()

Simple Animations

TransitionDrawable

A very simple animation can be implemented using TransitionDrawable. TransitionDrawable is an indirect subclass of Drawable. It provides a cross-fade between layers. It can be definable in XML and executed with startTransition(). For example, we define a TransitionDrawable in transitionimage.xml in the res/drawable/ folder as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<transition xmlns:android="http://schemas.android.com/apk/res/android">
       <item android:drawable="@drawable/chatimages1" />
       <item android:drawable="@drawable/chatimages" />
</transition>
```

It will cross-fade between these two images. We use this drawable as the image source of our image button in XML, and then call `startTransition(int)` to execute the simple animation, and pass in the length of the transition in milliseconds as the parameter.

```
<ImageButton
   android:id="@+id/imgBtnid"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:src="@drawable/transitionimage"
   android:adjustViewBounds="true"
   android:onClick="changeBackground"/>
```

```
val imageBtn = findViewById<ImageButton>(R.id.imgBtnid)
val drawable = imageBtn.drawable as TransitionDrawable
drawable.startTransition(3000)
```

AnimationDrawable

Similarly, we can create a frame animation by showing a sequence of images in order (like a film) with an AnimationDrawable. For example, res/drawable/animationlist.xml is defined as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android" android:oneshot
   <item android:drawable="@drawable/chatimages2p" android:duration="500" />
   <item android:drawable="@drawable/chatimages" android:duration="500" />
```

Then we use it just as other drawables. Here we add it dynamically in the code.

```
val animationIV = ImageView(this)
animationIV?.setBackgroundResource(R.drawable.animationlist)
val frameAnimation = animationIV!!.background as AnimationDrawable
```

Then we can call frameAnimation.start() to start the animation and frameAnimation.stop() to stop the animation. We can also use handlers to schedule them at a later time. The following addAnimation() method is called when the “add animation” button is clicked. It first creates a handler and schedules frameAnimation.start() after 1 second (1000 milliseconds) and then schedules frameAnimation.stop() after 6 seconds. So the animation will run for 5 seconds.

```
animationIV.setOnClickListener {
   // the animation will start after 1s
   val animationHandler = Handler(this.mainLooper)
   animationHandler.postDelayed({ frameAnimation!!.start() }, 1000)

   //the animation will stop after 6s
   animationHandler.postDelayed({ frameAnimation!!.stop() }, 6000)
}
```

Tweened Animation

Besides using Drawables, we can use Animation objects. Animation is an abstract class defined in the android.view.animation package for an Animation that can be applied to Views, Surfaces, or other objects. For example, we define The XML files in the res/anim/ define tween animations that will be handled by the android.view.animation package.The animation object can be referred via @[package:]anim/filename(in XML) in the XML layout, or R.anim.filename in Java. For example, we can perform a series of transformations on a single image with an Animation (such as fading, moving, stretching) and define the transformations in animateexample.xml as follows.

```
?xml version="1.0" encoding="utf-8"?>
>set xmlns:android="http://schemas.android.com/apk/res/android"
   android:shareInterpolator="false">
   <scale
       android:interpolator="@android:anim/accelerate_decelerate_interpolator"
       android:fromXScale="1.0"
       android:toXScale="1.4"
       android:fromYScale="1.0"
       android:toYScale="0.6"
       android:pivotX="50%"
       android:pivotY="50%"
```

```
        android:fillAfter="false"
        android:duration="2000" />
    <set
        android:interpolator="@android:anim/accelerate_interpolator"
        android:startOffset="700">
        <scale
            android:fromXScale="1.4"
            android:toXScale="0.0"
            android:fromYScale="0.6"
            android:toYScale="0.0"
            android:pivotX="50%"
            android:pivotY="50%"
            android:duration="3000" />
        <rotate
            android:fromDegrees="0"
            android:toDegrees="-45"
            android:toYScale="0.0"
            android:pivotX="50%"
            android:pivotY="50%"
            android:duration="4000" />
    </set>
</set>
```

Then we can load it in code using `AnimationUtils.loadAnimation()`. Then we can start this animation on the image view by calling its `startAnimation(Animation)` method.

```
findViewById<ImageView>(R.id.chatimageview).setOnClickListener{
    val hyperspaceJump = AnimationUtils.loadAnimation(this, R.anim.animateexample)
    it.startAnimation(hyperspaceJump)
}
```

For more information about the res/ folder structure, please read Providing alternative resources

Test Yourself

Where are tweened animation xml files stored?

res/anim/

Test Yourself

Where are frame animation xml files stored?

res/drawable

Test Yourself

How would you refer to /res/drawable/myanimation.xml in Java?

R.drawable.myanimation

Test Yourself

How would you define a frame animation?

Use <animation-list> tag, and specify each frame as an item.

Conclusion

In this module we show how to use pending intents to build notifications and set alarms. We also introduce how to use simple Android graphics APIs with image views and custom views, as well as how to create simple animations. In addition, we discuss several Android application security mechanisms and tips to make your app more secure.