|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Intent | <chare> | 1 | <pext> | An intent is a messaging object that is used  to request an action from other components of an application.  It can also be used to launch an activity, send SMS, send an email,  display a web page, etc.  It shows notification messages to the user.  It alerts the user of a particular state that occurred.  There are two types of intents in Android:  1. Implicit Intent- Used to invoke the system components.  2. Explicit Intent- Used to invoke the activity class. | </end> |
| <hitle> | Intent type - Implicit vs Explicit | <chare> | 2 | <pext> | 1. Explicit Intent:  An Explicit Intent is where you inform the system  about which activity should handle this intent.  Here target component is defined directly in the intent.  2. Implicit Intent:  An Implicit Intent permits you to declare the action you want to carry out.  Further, the Android system will check  which components are registered to handle that specific action  based on intent data.  Here target component is not defined in the intent. | </end> |
| <hitle> | Intent start activity | <chare> | 2 | <pext> | We 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.  If we want to receive a result from the activity when it finishes,  we can call startActivityForResult().  The activity receives the result as a separate Intent object  in the activity's onActivityResult() callback. | </end> |
| <hitle> | Intent Start service | <chare> | 2 | <pext> | We can start a service with JobScheduler.  For earlier versions, we can start a service by using methods  of the Service class.  We can start a service to perform a one-time operation,  like downloading a file by passing an Intent to startService().  The Intent describes the service to start and carries any necessary data.  If the service is designed with a client-server interface,  we can bind to the service from another component by passing  an Intent to bindService(). | </end> |
| <hitle> | Intent Deliver broadcast | <chare> | 2 | <pext> | The system delivers various broadcasts for system events,  such as when the system boots up or the device starts charging.  We can deliver a broadcast to other apps by passing an Intent  to sendBroadcast() or sendOrderedBroadcast(). | </end> |
| <hitle> | Build/make intent structure | <chare> | 2 | <pext> | 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.  Let’s see which information is contained in an Intent.  1. Component name  The name of the component to start.  2. Action  A string that specifies the generic action to perform such as view or pick.  Common actions for starting an activity are:  ACTION\_VIEW, ACTION\_SEND  3. Data  The URI that references the data  to be acted on and/or the MIME type of that data.  The type of data supplied is generally dictated by the intent's action  like, if the action is ACTION\_EDIT, the data should contain the URI  of the document to edit.  4. Category  A string containing additional information about the kind of component  that should handle the intent.  Common categories are  CATEGORY\_BROWSABLE, CATEGORY\_LAUNCHER  5. Extras  Key-value pairs that carry additional information required to accomplish  the requested action.  Just as some actions use particular kinds of data URIs,  some actions also use particular extras.  6. Flags  Flags are defined in the Intent class that function as metadata for the intent. | </end> |
| <hitle> | Intent Component name info | <chare> | 3 | <pext> | It is the name of the component to start.  This is optional, but it's the critical piece of information that makes  an intent explicit, meaning that the intent should be delivered only  to the app component defined by the component name.  Without a component name, the intent is implicit and the system decides  which component should receive the intent based  on the other intent information such as the action, data, and category.  If we need to start a specific component in the app,  We should specify the component name.  Note: When starting a Service, always specify the component name.  Otherwise, we cannot be certain what service will respond to the intent,  and the user cannot see which service starts.  This field of the Intent is a ComponentName object,  which we can specify using a fully qualified class name  of the target component, including the package name of the app,  for example, com.example.ExampleActivity.  We can set the component name with setComponent(), setClass(),  setClassName(), or with the Intent constructor. | </end> |
| <hitle> | Intent Action info | <chare> | 3 | <pext> | Intent Action is a string that specifies the generic action to perform  such as view or pick.  In the case of a broadcast intent, this is the action that took place  and is being reported.  The action largely determines how the rest of the intent is structured,  particularly the information that is contained in the data and extras.  We can specify actions for use by intents within the app,  or for use by other apps to invoke components in your app,  but we usually specify action constants defined by the Intent class  or other framework classes.  Common actions for starting an activity are:  1. ACTION\_VIEW  We can use this action in an intent with startActivity()  when you have some information that an activity can show to the user,  such as a photo to view in a gallery app, or an address to view in a map app.  2. ACTION\_SEND  Also known as the share intent, we should use this in an intent  with startActivity() when we have some data that the user can share  through another app, such as an email app or social sharing app.  Other actions are defined elsewhere in the Android framework,  such as in Settings for actions that open specific screens  in the system's Settings app.  We can specify the action for an intent with setAction()  or with an Intent constructor.  If you define our own actions, we need to be sure to include  app's package name as a prefix  const val ACTION\_TIMETRAVEL = "com.example.action.TIMETRAVEL" | </end> |
| <hitle> | Intent Data info | <chare> | 3 | <pext> | The URI that references the data to be acted on  and the MIME type of that data.  The type of data supplied is generally dictated by the intent's action.  For example, if the action is ACTION\_EDIT, the data should contain the URI  of the document to edit.  When creating an intent, it's often important to specify the type of data  in addition to its URI.  For example, an activity that's able to display images  probably won't be able to play an audio file,  even though the URI formats could be similar.  Specifying the MIME type of the data helps the Android system  find the best component to receive your intent.  However, the MIME type can sometimes be inferred from the URI,  particularly when the data is a content: URI.  A content: URI indicates the data is located on the device  and controlled by a ContentProvider, which makes the data MIME type  visible to the system.  To set only the data URI, call setData().  To set only the MIME type, call setType().  If necessary, we can set both explicitly with setDataAndType().  Caution: If we want to set both the URI and MIME type,  we don't call setData() and setType()  because they each nullify the value of the other.  Always use setDataAndType() to set both URI and MIME type. | </end> |
| <hitle> | Intent Category info | <chare> | 3 | <pext> | It is a string containing additional information about the kind  of component that should handle the intent.  Any number of category descriptions can be placed in an intent,  but most intents do not require a category.  Common categories are:  1. CATEGORY\_BROWSABLE  The target activity allows itself to be started by a web browser  to display data referenced by a link,  such as an image or an e-mail message.  2. CATEGORY\_LAUNCHER  The activity is the initial activity of a task and is listed  in the system's application launcher.  We can specify a category with addCategory(). | </end> |
| <hitle> | Intent Extra info | <chare> | 3 | <pext> | It’s Key-value pairs that carry additional information required  to accomplish the requested action.  Just as some actions use particular kinds of data URIs,  some actions also use particular extras.  We can add extra data with various putExtra() methods,  each accepting two parameters: the key name and the value.  We can also create a Bundle object with all the extra data,  then insert the Bundle in the Intent with putExtras().  For example, when creating an intent to send an email with ACTION\_SEND,  we can specify the to recipient with the EXTRA\_EMAIL key,  and specify the subject with the EXTRA\_SUBJECT key.  The Intent class specifies many EXTRA\_\* constants  for standardized data types.  If we need to declare our own extra keys for intents that the app receives,  we need to be sure to include app's package name as a prefix  const val EXTRA\_GIGAWATTS = "com.example.EXTRA\_GIGAWATTS"  Caution: We shoud not use Parcelable or Serializable data  when sending an intent that we expect another app to receive.  If an app attempts to access data in a Bundle object  but does not have access to the parceled or serialized class,  the system raises a RuntimeException. | </end> |
| <hitle> | Intent Flags Info | <chare> | 3 | <pext> | Flags are defined in the Intent class that function  as metadata for the intent.  The flags may instruct the Android system how to launch an activity  for example, which task the activity should belong to  and how to treat it after it's launched for example,  whether it belongs in the list of recent activities.  We can set flags by setFlags() method. | </end> |
| <hitle> | howto explicit intent download Service | <chare> | 2 | <pext> | An explicit intent is one that we use to launch a specific app component,  such as a particular activity or service in your app.  To create an explicit intent, we need to define the component name  for the Intent object, and all other intent properties are optional.  For example, if we built a service in the app, named DownloadService,  designed to download a file from the web,  we can do like this.  // Executed in an Activity, so 'this' is the Context  // The fileUrl is a string URL, such as "http://www.example.com/image.png"  val downloadIntent = Intent(this, DownloadService::class.java).apply {  data = 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. | </end> |
| <hitle> | howto implicit intent Share Action | <chare> | 2 | <pext> | An implicit intent specifies an action that can invoke  any app on the device able to perform the action.  Using an implicit intent is useful when the 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 we have content that we want the user to share  with other people, we can create an intent with the ACTION\_SEND action  and add extras that specify the content to share.  When we call startActivity() with that intent,  the user can pick an app through which to share the content.  // Create the text message with a string.  val sendIntent = Intent().apply {  action = Intent.ACTION\_SEND  putExtra(Intent.EXTRA\_TEXT, textMessage)  type = "text/plain"  }  // Try to invoke the intent.  try {  startActivity(sendIntent)  } catch (e: ActivityNotFoundException) {  // Define what your app should do if no activity can handle the intent.  }  When startActivity() is called, the system examines  all of the installed apps to determine which ones can handle  this kind of intent.  If there's only one app that can handle it, that app opens immediately  and is given the intent.  If no other apps can handle it, the app can catch  the ActivityNotFoundException that occurs.  If multiple activities accept the intent, the system displays a dialog  so the user can pick which app to use. | </end> |
| <hitle> | app chooser intent createChooser() | <chare> | 2 | <pext> | To show the chooser, we can create an Intent using createChooser()  and pass it to startActivity().  val sendIntent = Intent(Intent.ACTION\_SEND)  ...  // Always use string resources for UI text.  // This says something like "Share this photo with"  val title: String = resources.getString(R.string.chooser\_title)  // Create intent to show the chooser dialog  val chooser: Intent = Intent.createChooser(sendIntent, title)  // Verify the original intent will resolve to at least one activity  if (sendIntent.resolveActivity(packageManager) != null) {  startActivity(chooser)  }  (\*It shows a dialog with a list of apps that respond to the intent passed to the createChooser() method and uses the supplied text as the dialog title.\*) | </end> |
| <hitle> | unsafe intent launch | <chare> | 2 | <pext> | The app might launch intents to navigate between components  inside of your app, or to perform an action on behalf of another app.  To improve platform security, It provides a debugging feature  that warns us if the app performs an unsafe launch of an intent.  For example, the app might perform an unsafe launch of a nested intent,  which is an intent that is passed as an extra in another intent.  If the app performs both of the following actions,  the system detects an unsafe intent launch,  and a StrictMode violation occurs:  1. The app unparcels a nested intent from the extras  of a delivered intent.  2. The app immediately starts an app component using that nested intent,  such as passing the intent into startActivity(), startService(),  or bindService(). | </end> |
| <hitle> | Check unsafe intent launch | <chare> | 2 | <pext> | To check for unsafe intent launches in the app,  we can call detectUnsafeIntentLaunch() when we configure VmPolicy.  If the app detects a StrictMode violation, we might want  to stop app execution to protect potentially sensitive information.  Note: If the app targets Android 12 and uses the detectAll() method  in its VmPolicy definition, the detectUnsafeIntentLaunch() method  is called automatically.  fun onCreate() {  StrictMode.setVmPolicy(VmPolicy.Builder()  // Other StrictMode checks that you've previously added.  // ...  .detectUnsafeIntentLaunch()  .penaltyLog()  // Consider also adding penaltyDeath()  .build())  } | </end> |
| <hitle> | PendingIntent | <chare> | 2 | <pext> | PendingIntent is a high-level API that lets you create and manage  asynchronous requests to other apps.  You can use this class for starting activities, delivering results,  and receiving results from other apps.  PendingIntent starts at some point in the future but Intent starts immediately.  Pending Intent is frequently used in:  1. NotificationiManager  2. AlarmManager  3. AppWidgetManager  how the system passes control from your client app to another service app, and back to your app?  1. The app creates an intent that invokes an activity in another app.  Within that intent, we add a PendingIntent object as an extra.  This pending intent invokes a component in the app and  this component isn't exported.  2. Upon receiving the app's intent, the other app extracts  the nested PendingIntent object.  3. The other app invokes the send() method  on the PendingIntent object.  4. After passing control back to your app,  the system invokes the pending intent using your app's context. | </end> |
| <hitle> | Intent filter | <chare> | 2 | <pext> | To advertise which implicit intents the app can receive,  we can declare one or more intent filters  for each of the app components with an <intent-filter> element  in manifest file.  Each intent filter specifies the type of intents it accepts based on  the intent's action, data, and category.  The system delivers an implicit intent to the app component  only if the intent can pass through one of your intent filters.  Note: An explicit intent is always delivered to its target,  regardless of any intent filters the component declares.  An app component should declare separate filters  for each unique job it can do.  For example, one activity in an image gallery app may have two filters:  one filter to view an image, and another filter to edit an image.  When the activity starts, it inspects the Intent and decides  how to behave based on the information in the Intent  such as to show the editor controls or not.  Each intent filter is defined by an <intent-filter> element  in the app's manifest file,  nested in the corresponding app component  such as an <activity> element.  In each app component that includes an <intent-filter> element,  explicitly we can set a value for android:exported.  This attribute indicates whether the app component  is accessible to other apps.  In some situations, such as activities whose intent filters  include the LAUNCHER category, it's useful to set this attribute to true.  Otherwise, it's safer to set this attribute to false.  Inside the <intent-filter>, we can specify the type of intents  to accept using one or more of <action>, <data>, <category>.  <activity android:name="ShareActivity" android:exported="false">  <intent-filter>  <action android:name="android.intent.action.SEND"/>  <category android:name="android.intent.category.DEFAULT"/>  <data android:mimeType="text/plain"/>  </intent-filter>  </activity>  (\*For example, an activity declaration with an intent filter  to receive an ACTION\_SEND intent when the data type is text.\*)  When we want to handle multiple kinds of intents,  but only in specific combinations of action, data, and category type,  then we need to create multiple intent filters. | </end> |
| <hitle> | social-sharing Intent filter | <chare> | 2 | <pext> | <activity android:name="MainActivity" android:exported="true">  <!-- This activity is the main entry, should appear in app launcher -->  <intent-filter>  <action android:name="android.intent.action.MAIN" />  <category android:name="android.intent.category.LAUNCHER" />  </intent-filter>  </activity>  <activity android:name="ShareActivity" android:exported="false">  <!-- This activity handles "SEND" actions with text data -->  <intent-filter>  <action android:name="android.intent.action.SEND"/>  <category android:name="android.intent.category.DEFAULT"/>  <data android:mimeType="text/plain"/>  </intent-filter>  <!-- This activity also handles "SEND" and "SEND\_MULTIPLE" with media data -->  <intent-filter>  <action android:name="android.intent.action.SEND"/>  <action android:name="android.intent.action.SEND\_MULTIPLE"/>  <category android:name="android.intent.category.DEFAULT"/>  <data android:mimeType="application/vnd.google.panorama360+jpg"/>  <data android:mimeType="image/\*"/>  <data android:mimeType="video/\*"/>  </intent-filter>  </activity>  The first activity, MainActivity, is the app's main entry point  and the activity that opens when the user initially launches the app  with the launcher icon:  1. The ACTION\_MAIN action indicates this is the main entry point  and does not expect any intent data.  2. The CATEGORY\_LAUNCHER category indicates that this activity's icon  should be placed in the system's app launcher.  If the <activity> element does not specify an icon with icon,  then the system uses the icon from the <application> element.  These two must be paired together in order for the activity to appear  in the app launcher.  The second activity, ShareActivity, is intended to facilitate sharing text  and media content.  Although users might enter this activity by navigating to it  from MainActivity, they can also enter ShareActivity directly  from another app that issues an implicit intent  matching one of the two intent filters. | </end> |
| <hitle> | pending intent | <chare> | 2 | <pext> | UpdatingRecords  UpdatingRecords  UpdatingRecords  UpdatingRecords  UpdatingRecords | </end> |
| <hitle> | Alarm Clock Intent | <chare> | 2 | <pext> | ---------------Create an alarm---------------------- Intent  To create a new alarm, we can use the ACTION\_SET\_ALARM action  and specify alarm details such as the time and message using extras.  Extras are:  1. Action: ACTION\_SET\_ALARM  2. Data URI: None  3. MIME Type: None  4. Extras:  EXTRA\_HOUR - The hour for the alarm.  EXTRA\_MINUTES - The minutes for the alarm.  EXTRA\_MESSAGE - A custom message to identify the alarm.  EXTRA\_DAYS - An ArrayList including each week day.  EXTRA\_RINGTONE, EXTRA\_VIBRATE, EXTRA\_SKIP\_UI  fun createAlarm(message: String, hour: Int, minutes: Int) {  val intent = Intent(AlarmClock.ACTION\_SET\_ALARM).apply {  putExtra(AlarmClock.EXTRA\_MESSAGE, message)  putExtra(AlarmClock.EXTRA\_HOUR, hour)  putExtra(AlarmClock.EXTRA\_MINUTES, minutes)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <uses-permission android:name="com.android.alarm.permission.SET\_ALARM" />  (\*permission\*)  <activity ...>  <intent-filter>  <action android:name="android.intent.action.SET\_ALARM" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity>  (\*Intent-filter\*)  ---------------Create a timer---------------------- Intent  To create a countdown timer, we can use the ACTION\_SET\_TIMER action  and specify timer details such as the duration.  1. Action: ACTION\_SET\_TIMER  2. Data URI: None  3. MIME Type: None  4. Extras:  EXTRA\_LENGTH - The length of the timer in seconds.  EXTRA\_MESSAGE - A custom message to identify the timer.  EXTRA\_SKIP\_UI - A boolean specifying whether the responding app should skip its UI  fun startTimer(message: String, seconds: Int) {  val intent = Intent(AlarmClock.ACTION\_SET\_TIMER).apply {  putExtra(AlarmClock.EXTRA\_MESSAGE, message)  putExtra(AlarmClock.EXTRA\_LENGTH, seconds)  putExtra(AlarmClock.EXTRA\_SKIP\_UI, true)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <uses-permission android:name="com.android.alarm.permission.SET\_ALARM" />  <activity ...>  <intent-filter>  <action android:name="android.intent.action.SET\_TIMER" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity>  ------------------Show all alarms---------------------- Intent  To show the list of alarms, we can use the ACTION\_SHOW\_ALARMS action.  1. Action: ACTION\_SHOW\_ALARMS  2. Data URI: None  3. MIME Type: None  <activity ...>  <intent-filter>  <action android:name="android.intent.action.SHOW\_ALARMS" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity> | </end> |
| <hitle> | Calendar Intent | <chare> | 2 | <pext> | -----------Add a calendar event-------------- Intent  To add a new event to the user's calendar, we can use  the ACTION\_INSERT action and specify the data URI  with Events.CONTENT\_URI.  1. Action: ACTION\_INSERT  2. Data URI: Events.CONTENT\_URI  3. MIME Type: "vnd.android.cursor.dir/event"  4. Extras:  EXTRA\_EVENT\_ALL\_DAY - A boolean specifying whether this is an all-day event.  EXTRA\_EVENT\_BEGIN\_TIME - The start time of the event.  EXTRA\_EVENT\_END\_TIME - The end time of the event.  TITLE - The event title.  DESCRIPTION - The event description.  EVENT\_LOCATION - The event location.  EXTRA\_EMAIL - A comma-separated list of email addresses that specify the invitees.  fun addEvent(title: String, location: String, begin: Long, end: Long) {  val intent = Intent(Intent.ACTION\_INSERT).apply {  data = Events.CONTENT\_URI  putExtra(Events.TITLE, title)  putExtra(Events.EVENT\_LOCATION, location)  putExtra(CalendarContract.EXTRA\_EVENT\_BEGIN\_TIME, begin)  putExtra(CalendarContract.EXTRA\_EVENT\_END\_TIME, end)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <activity ...>  <intent-filter>  <action android:name="android.intent.action.INSERT" />  <data android:mimeType="vnd.android.cursor.dir/event" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity> | </end> |
| <hitle> | Camera Intent | <chare> | 2 | <pext> | -----------Capture a picture or video and return it---------------  To open a camera app and receive the resulting photo or video,  we can use the ACTION\_IMAGE\_CAPTURE  or ACTION\_VIDEO\_CAPTURE action.  Also we can specify the URI location where the camera to save  the photo or video, in the EXTRA\_OUTPUT extra.  1. Action:  ACTION\_IMAGE\_CAPTURE or ACTION\_VIDEO\_CAPTURE  2. Data URI Scheme: None  3. MIME Type: None  4. Extras:  EXTRA\_OUTPUT - The URI location where the camera app should save.  Note: When you use ACTION\_IMAGE\_CAPTURE to capture a photo, the camera may also return a downscaled copy (a thumbnail) of the photo in the result Intent, saved as a Bitmap in an extra field named "data".  val REQUEST\_IMAGE\_CAPTURE = 1  private fun dispatchTakePictureIntent() {  val takePictureIntent = Intent(MediaStore.ACTION\_IMAGE\_CAPTURE)  try {  startActivityForResult(takePictureIntent, REQUEST\_IMAGE\_CAPTURE)  } catch (e: ActivityNotFoundException) {  // display error state to the user  }  }  <activity ...>  <intent-filter>  <action android:name="android.media.action.IMAGE\_CAPTURE" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity>  --------------Start a camera app in still image mode---------  To open a camera app in still image mode, we can use  the INTENT\_ACTION\_STILL\_IMAGE\_CAMERA action.  1. Action: INTENT\_ACTION\_STILL\_IMAGE\_CAMERA  2. Data URI Scheme: None  3. MIME Type: None  4. Extras: None  private fun dispatchTakePictureIntent() {  val takePictureIntent = Intent(MediaStore.ACTION\_IMAGE\_CAPTURE)  try {  startActivityForResult(takePictureIntent, REQUEST\_IMAGE\_CAPTURE)  } catch (e: ActivityNotFoundException) {  // display error state to the user  }  }  <activity ...>  <intent-filter>  <action android:name="android.media.action.STILL\_IMAGE\_CAMERA" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity>  ----------Start a camera app in video mode----------------  To open a camera app in video mode, we can use  the INTENT\_ACTION\_VIDEO\_CAMERA action.  1. Action: INTENT\_ACTION\_VIDEO\_CAMERA  2. Data URI Scheme: None  3. MIME Type: None  4. Extras: None  fun capturePhoto() {  val intent = Intent(MediaStore.INTENT\_ACTION\_VIDEO\_CAMERA)  if (intent.resolveActivity(packageManager) != null) {  startActivityForResult(intent, REQUEST\_IMAGE\_CAPTURE)  }  }  <activity ...>  <intent-filter>  <action android:name="android.media.action.VIDEO\_CAMERA" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity> | </end> |
| <hitle> | Contacts/People App Intent | <chare> | 2 | <pext> | -----------Select a contact----------------  To have the user select a contact and provide the app access  to all the contact information, we can use the ACTION\_PICK action  and specify the MIME type to Contacts.CONTENT\_TYPE.  The result Intent delivered to onActivityResult() callback contains  the content: URI pointing to the selected contact.  The response grants the app temporary permissions to read that contact  using the Contacts Provider API even if the app does not include  the READ\_CONTACTS permission.  1. Action: ACTION\_PICK  2. Data URI Scheme: None  3. MIME Type: Contacts.CONTENT\_TYPE  const val REQUEST\_SELECT\_CONTACT = 1  fun selectContact() {  val intent = Intent(Intent.ACTION\_PICK).apply {  type = ContactsContract.Contacts.CONTENT\_TYPE  }  if (intent.resolveActivity(packageManager) != null) {  startActivityForResult(intent, REQUEST\_SELECT\_CONTACT)  }  }  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent) {  if (requestCode == REQUEST\_SELECT\_CONTACT && resultCode == RESULT\_OK) {  val contactUri: Uri = data.data  // Do something with the selected contact at contactUri  //...  }  }  ---------Select specific contact data------------------  To have the user select a specific piece of information from a contact,  such as a phone number, email address, or other data type,  we can use the ACTION\_PICK action and specify the MIME type  to one of the content types,  such as CommonDataKinds.Phone.CONTENT\_TYPE  to get the contact's phone number.  In many cases, the app needs to have the READ\_CONTACTS permission  in order to view specific information about a particular contact.  If you need to retrieve only one type of data from a contact,  this technique with a CONTENT\_TYPE  from the ContactsContract.CommonDataKinds classes  is more efficient than using the Contacts.CONTENT\_TYPE  because the result provides you direct access to the desired data  without requiring you to perform a more complex query to Contacts Provider.  The result Intent delivered to your onActivityResult() callback contains  the content: URI pointing to the selected contact data.  The response grants the app temporary permissions  to read that contact data even if your app does not include  the READ\_CONTACTS permission.  1. Action: ACTION\_PICK  2. Data URI Scheme: None  3. MIME Type:  CommonDataKinds.Phone.CONTENT\_TYPE - Pick from contacts with a phone number.  CommonDataKinds.Email.CONTENT\_TYPE - Pick from contacts with an email address.  CommonDataKinds.StructuredPostal.CONTENT\_TYPE - Pick from contacts with a postal address.  const val REQUEST\_SELECT\_PHONE\_NUMBER = 1  fun selectContact() {  // Start an activity for the user to pick a phone number from contacts  val intent = Intent(Intent.ACTION\_PICK).apply {  type = CommonDataKinds.Phone.CONTENT\_TYPE  }  if (intent.resolveActivity(packageManager) != null) {  startActivityForResult(intent, REQUEST\_SELECT\_PHONE\_NUMBER)  }  }  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent) {  if (requestCode == REQUEST\_SELECT\_PHONE\_NUMBER && resultCode == Activity.RESULT\_OK) {  // Get the URI and query the content provider for the phone number  val contactUri: Uri = data.data  val projection: Array<String> = arrayOf(CommonDataKinds.Phone.NUMBER)  contentResolver.query(contactUri, projection, null, null, null).use { cursor ->  // If the cursor returned is valid, get the phone number  if (cursor.moveToFirst()) {  val numberIndex = cursor.getColumnIndex(CommonDataKinds.Phone.NUMBER)  val number = cursor.getString(numberIndex)  // Do something with the phone number  ...  }  }  }  }  ---------------View a contact-----------------------  To display the details for a known contact, we can use  the ACTION\_VIEW action and specify the contact  with a content: URI as the intent data.  1. Action: ACTION\_VIEW  2. Data URI Scheme: content:<URI>  3. MIME Type: None. The type is inferred from contact URI.  There are primarily two ways to initially retrieve the contact's URI:  1. We can use the contact URI returned by the ACTION\_PICK.  2. We can access the list of all contacts directly.  fun viewContact(contactUri: Uri) {  val intent = Intent(Intent.ACTION\_VIEW, contactUri)  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  -----------Edit an existing contact-----------------  To edit a known contact, we can use the ACTION\_EDIT action,  specify the contact with a content: URI as the intent data,  and include any known contact information in extras specified  by constants in ContactsContract.Intents.Insert.  1. Action: ACTION\_EDIT  2. Data URI Scheme: content:<URI>  3. MIME Type: The type is inferred from contact URI.  4. Extras: One or more of the extras defined in ContactsContract.Intents.Insert so we can populate fields of the contact details.  There are primarily two ways to initially retrieve the contact's URI:  1. We can use the contact URI returned by the ACTION\_PICK.  2. We can access the list of all contacts directly.  fun editContact(contactUri: Uri, email: String) {  val intent = Intent(Intent.ACTION\_EDIT).apply {  data = contactUri  putExtra(ContactsContract.Intents.Insert.EMAIL, email)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  ----------------Insert a contact-----------------  To insert a new contact, we can use the ACTION\_INSERT action,  specify Contacts.CONTENT\_TYPE as the MIME type, and include  any known contact information in extras specified by constants  in ContactsContract.Intents.Insert.  1. Action: ACTION\_INSERT  2. Data URI Scheme: None  3. MIME Type: Contacts.CONTENT\_TYPE  4. Extras: One or more of the extras defined in ContactsContract.Intents.Insert.  fun insertContact(name: String, email: String) {  val intent = Intent(Intent.ACTION\_INSERT).apply {  type = ContactsContract.Contacts.CONTENT\_TYPE  putExtra(ContactsContract.Intents.Insert.NAME, name)  putExtra(ContactsContract.Intents.Insert.EMAIL, email)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  } | </end> |
| <hitle> | Email Intent | <chare> | 2 | <pext> | ------------Compose an email with optional attachments--------  1. Action:  ACTION\_SENDTO (for no attachment) or  ACTION\_SEND (for one attachment) or  ACTION\_SEND\_MULTIPLE (for multiple attachments)  2. Data URI Scheme: None  MIME Type: "text/plain", "\*/\*"  Extras:  Intent.EXTRA\_EMAIL, Intent.EXTRA\_CC, Intent.EXTRA\_BCC,  Intent.EXTRA\_SUBJECT, Intent.EXTRA\_TEXT,  Intent.EXTRA\_STREAM (A Uri pointing to the attachment. If using the ACTION\_SEND\_MULTIPLE action, this should instead be an ArrayList containing multiple Uri objects.)  fun composeEmail(addresses: Array<String>, subject: String, attachment: Uri) {  val intent = Intent(Intent.ACTION\_SEND).apply {  type = "\*/\*"  putExtra(Intent.EXTRA\_EMAIL, addresses)  putExtra(Intent.EXTRA\_SUBJECT, subject)  putExtra(Intent.EXTRA\_STREAM, attachment)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  If you want to ensure that the intent is handled only by an email app and not other text messaging or social apps, then we can use the ACTION\_SENDTO action and include the "mailto:" data scheme.  fun composeEmail(addresses: Array<String>, subject: String) {  val intent = Intent(Intent.ACTION\_SENDTO).apply {  data = Uri.parse("mailto:") // only email apps should handle this  putExtra(Intent.EXTRA\_EMAIL, addresses)  putExtra(Intent.EXTRA\_SUBJECT, subject)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <activity ...>  <intent-filter>  <action android:name="android.intent.action.SEND" />  <data android:type="\*/\*" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  <intent-filter>  <action android:name="android.intent.action.SENDTO" />  <data android:scheme="mailto" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity> | </end> |
| <hitle> | File Storage Intent | <chare> | 2 | <pext> | -------------Retrieve a specific type of file--------------  We can use the ACTION\_GET\_CONTENT action and specify your desired MIME type.  The file reference returned to your app is transient  to the activity's current lifecycle, so if we want to access it later  we must import a copy that you can read later.  This intent also allows the user to create a new file in the process  for example, instead of selecting an existing photo,  the user can capture a new photo with the camera.  The result intent delivered to the onActivityResult() method  includes data with a URI pointing to the file.  However, if we'd like to restrict selectable files to only those  that are accessible from a content provider  and that are available as a file stream with openFileDescriptor(),  we should add the CATEGORY\_OPENABLE category to your intent.  We can also allow the user to select multiple files  by adding EXTRA\_ALLOW\_MULTIPLE to the intent, set to true.  We can then access each of the selected files in a ClipData object returned by getClipData().  1. Action: ACTION\_GET\_CONTENT  2. Data URI Scheme: None  3. MIME Type: The MIME type corresponding to the file type the user should select.  4. Extras:  EXTRA\_ALLOW\_MULTIPLE - A boolean declaring whether the user can select more than one file at a time.  EXTRA\_LOCAL\_ONLY - A boolean that declares whether the returned file must be available directly from the device, rather than requiring a download from a remote service.  5. Category (optional):  CATEGORY\_OPENABLE - To return only "openable" files that can be represented as a file stream with openFileDescriptor().  const val REQUEST\_IMAGE\_GET = 1  fun selectImage() {  val intent = Intent(Intent.ACTION\_GET\_CONTENT).apply {  type = "image/\*"  }  if (intent.resolveActivity(packageManager) != null) {  startActivityForResult(intent, REQUEST\_IMAGE\_GET)  }  }  (\*Intent to get a photo\*)  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent) {  if (requestCode == REQUEST\_IMAGE\_GET && resultCode == Activity.RESULT\_OK) {  val thumbnail: Bitmap = data.getParcelableExtra("data")  val fullPhotoUri: Uri = data.data  // Do work with photo saved at fullPhotoUri  ...  }  }  <activity ...>  <intent-filter>  <action android:name="android.intent.action.GET\_CONTENT" />  <data android:type="image/\*" />  <category android:name="android.intent.category.DEFAULT" />  <!-- The OPENABLE category declares that the returned file is accessible  from a content provider that supports OpenableColumns  and ContentResolver.openFileDescriptor() -->  <category android:name="android.intent.category.OPENABLE" />  </intent-filter>  </activity>  (\*Intent filter to return a photo\*)  ----------------Open a specific type of file--------------------  We can request to open a file that's managed by another app  by using the ACTION\_OPEN\_DOCUMENT action and specifying a MIME type.  To also allow the user to instead create a new document  that your app can write to, we can use  the ACTION\_CREATE\_DOCUMENT action instead.  Whereas the intent delivered to your onActivityResult() method  from the ACTION\_GET\_CONTENT action may return a URI of any type,  the result intent from ACTION\_OPEN\_DOCUMENT  and ACTION\_CREATE\_DOCUMENT always specify the chosen file  as a content: URI that's backed by a DocumentsProvider.  We can open the file with openFileDescriptor() and query its details  using columns from DocumentsContract.Document.  The returned URI grants the app long-term read access to the file  also possibly with write access.  We can also allow the user to select multiple files by adding  EXTRA\_ALLOW\_MULTIPLE to the intent, set to true.  If the user selects just one item,  then we can retrieve the item from getData().  If the user selects more than one item, then getData() returns null  and we must instead retrieve each item from a ClipData object  that is returned by getClipData().  1. Action:  ACTION\_OPEN\_DOCUMENT or  ACTION\_CREATE\_DOCUMENT  2. Data URI Scheme: None  3: MIME Type: The MIME type corresponding to the file type the user should select.  4. Extras  EXTRA\_MIME\_TYPES,  EXTRA\_ALLOW\_MULTIPLE,  EXTRA\_TITLE,  EXTRA\_LOCAL\_ONLY,  A boolean that declares whether the returned file must be available directly from the device, rather than requiring a download from a remote service.  5. Category: CATEGORY\_OPENABLE  const val REQUEST\_IMAGE\_OPEN = 1  fun selectImage2() {  val intent = Intent(Intent.ACTION\_OPEN\_DOCUMENT).apply {  type = "image/\*"  addCategory(Intent.CATEGORY\_OPENABLE)  }  // Only the system receives the ACTION\_OPEN\_DOCUMENT, so no need to test.  startActivityForResult(intent, REQUEST\_IMAGE\_OPEN)  }  (\*Example intent to get a photo\*)  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent) {  if (requestCode == REQUEST\_IMAGE\_OPEN && resultCode == Activity.RESULT\_OK) {  val fullPhotoUri: Uri = data.data  // Do work with full size photo saved at fullPhotoUri  ...  }  }  Third party apps cannot actually respond to an intent  with the ACTION\_OPEN\_DOCUMENT action.  Instead, the system receives this intent and displays all the files  available from various apps in a unified user interface.  To provide the app's files in this UI and allow other apps to open them,  we must implement a DocumentsProvider and include an intent filter  for PROVIDER\_INTERFACE ("android.content.action.DOCUMENTS\_PROVIDER").  <provider ...  android:grantUriPermissions="true"  android:exported="true"  android:permission="android.permission.MANAGE\_DOCUMENTS">  <intent-filter>  <action android:name="android.content.action.DOCUMENTS\_PROVIDER" />  </intent-filter>  </provider> | </end> |
| <hitle> | Call a car Intent | <chare> | 2 | <pext> | To call a taxi, we can use  the ACTION\_RESERVE\_TAXI\_RESERVATION action.  1. Action: ACTION\_RESERVE\_TAXI\_RESERVATION  2. Data URI: None  3. MIME Type: None  4. Extras: None  fun callCar() {  val intent = Intent(ReserveIntents.ACTION\_RESERVE\_TAXI\_RESERVATION)  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <activity ...>  <intent-filter>  <action android:name="com.google.android.gms.actions.RESERVE\_TAXI\_RESERVATION" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity> | </end> |
| <hitle> | Maps Intent | <chare> | 2 | <pext> | ------Show a location on a map-----  To open a map, we can use the ACTION\_VIEW action  and specify the location information in the intent data  with one of the schemes  1. Action: ACTION\_VIEW  2. Data URI Scheme:  geo:latitude,longitude - Show the map at the given longitude and latitude.  geo:latitude,longitude?z=zoom - Show the map at the given longitude and latitude at a certain zoom level.  geo:0,0?q=lat,lng(label) - Show the map at the given longitude and latitude with a string label.  geo:0,0?q=my+street+address - Show the location for "my street address" (may be a specific address or location query).  3. MIME Type: None  fun showMap(geoLocation: Uri) {  val intent = Intent(Intent.ACTION\_VIEW).apply {  data = geoLocation  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <activity ...>  <intent-filter>  <action android:name="android.intent.action.VIEW" />  <data android:scheme="geo" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity> | </end> |
| <hitle> | Music or Video Intent | <chare> | 2 | <pext> | --------------Play a media file----------------------  To play a music file, we can use the ACTION\_VIEW action  and specify the URI location of the file in the intent data.  1. Action: ACTION\_VIEW  2. Data URI Scheme: file:<URI>, content:<URI>, http:<URL>  3. MIME Type: "audio/\*", "application/ogg", "application/x-ogg", "application/itunes"  Or any other that your app may require.  fun playMedia(file: Uri) {  val intent = Intent(Intent.ACTION\_VIEW).apply {  data = file  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <activity ...>  <intent-filter>  <action android:name="android.intent.action.VIEW" />  <data android:type="audio/\*" />  <data android:type="application/ogg" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity>  --------------Play music based on a search query-----------------  To play music based on a search query, we can use  the INTENT\_ACTION\_MEDIA\_PLAY\_FROM\_SEARCH intent.  This intent should include the EXTRA\_MEDIA\_FOCUS string extra,  which specifies the intended search mode.  For example, the search mode can specify whether the search  is for an artist name or song name.  1. Action: INTENT\_ACTION\_MEDIA\_PLAY\_FROM\_SEARCH  2. Data URI Scheme: None  3. MIME Type: None  4. Extras:  MediaStore.EXTRA\_MEDIA\_FOCUS (required) - Indicates the search mode whether the user is looking for a particular artist, album, song, or playlist.  Intent music search modes take additional extras.  search modes for each value of EXTRA\_MEDIA\_FOCUS are:  1. Any - "vnd.android.cursor.item/\*",  2. Unstructured - "vnd.android.cursor.item/\*"  3. Genre - Audio.Genres.ENTRY\_CONTENT\_TYPE  4. Artist - Audio.Artists.ENTRY\_CONTENT\_TYPE  5. Album - Audio.Albums.ENTRY\_CONTENT\_TYPE  6. Song - "vnd.android.cursor.item/audio"  7. Playlist - Audio.Playlists.ENTRY\_CONTENT\_TYPE  fun playSearchArtist(artist: String) {  val intent = Intent(MediaStore.INTENT\_ACTION\_MEDIA\_PLAY\_FROM\_SEARCH).apply {  putExtra(MediaStore.EXTRA\_MEDIA\_FOCUS, MediaStore.Audio.Artists.ENTRY\_CONTENT\_TYPE)  putExtra(MediaStore.EXTRA\_MEDIA\_ARTIST, artist)  putExtra(SearchManager.QUERY, artist)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <activity ...>  <intent-filter>  <action android:name="android.media.action.MEDIA\_PLAY\_FROM\_SEARCH" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity>  When handling this intent, the activity should check the value  of the EXTRA\_MEDIA\_FOCUS extra in the incoming Intent  to determine the search mode.  Once the activity has identified the search mode, it should read the values  of the additional extras for that particular search mode.  With this information the app can then perform the search  within its inventory to play the content that matches the search query.  override fun onCreate(savedInstanceState: Bundle?) {  ...  if (intent.action.compareTo(MediaStore.INTENT\_ACTION\_MEDIA\_PLAY\_FROM\_SEARCH) == 0) {  val mediaFocus: String? = intent.getStringExtra(MediaStore.EXTRA\_MEDIA\_FOCUS)  val query: String? = intent.getStringExtra(SearchManager.QUERY)  // Some of these extras may not be available depending on the search mode  val album: String? = intent.getStringExtra(MediaStore.EXTRA\_MEDIA\_ALBUM)  val artist: String? = intent.getStringExtra(MediaStore.EXTRA\_MEDIA\_ARTIST)  val genre: String? = intent.getStringExtra("android.intent.extra.genre")  val playlist: String? = intent.getStringExtra("android.intent.extra.playlist")  val title: String? = intent.getStringExtra(MediaStore.EXTRA\_MEDIA\_TITLE)  // Determine the search mode and use the corresponding extras  when {  mediaFocus == null -> {  // 'Unstructured' search mode (backward compatible)  playUnstructuredSearch(query)  }  mediaFocus.compareTo("vnd.android.cursor.item/\*") == 0 -> {  if (query?.isNotEmpty() == true) {  // 'Unstructured' search mode  playUnstructuredSearch(query)  } else {  // 'Any' search mode  playResumeLastPlaylist()  }  }  mediaFocus.compareTo(MediaStore.Audio.Genres.ENTRY\_CONTENT\_TYPE) == 0 -> {  // 'Genre' search mode  playGenre(genre)  }  mediaFocus.compareTo(MediaStore.Audio.Artists.ENTRY\_CONTENT\_TYPE) == 0 -> {  // 'Artist' search mode  playArtist(artist, genre)  }  mediaFocus.compareTo(MediaStore.Audio.Albums.ENTRY\_CONTENT\_TYPE) == 0 -> {  // 'Album' search mode  playAlbum(album, artist)  }  mediaFocus.compareTo("vnd.android.cursor.item/audio") == 0 -> {  // 'Song' search mode  playSong(album, artist, genre, title)  }  mediaFocus.compareTo(MediaStore.Audio.Playlists.ENTRY\_CONTENT\_TYPE) == 0 -> {  // 'Playlist' search mode  playPlaylist(album, artist, genre, playlist, title)  }  }  }  } | </end> |
| <hitle> | Note Intent | <chare> | 2 | <pext> | ------------------Create a note-------------------  To create a new note, we can use the ACTION\_CREATE\_NOTE action  and specify note details such as the subject and text using extras.  1. Action: ACTION\_CREATE\_NOTE  2. Data URI Scheme: None  3. MIME Type: PLAIN\_TEXT\_TYPE, "\*/\*"  4. Extras:  EXTRA\_NAME -A string indicating the title or subject of the note.  EXTRA\_TEXT - A string indicating the text of the note.  fun createNote(subject: String, text: String) {  val intent = Intent(NoteIntents.ACTION\_CREATE\_NOTE).apply {  putExtra(NoteIntents.EXTRA\_NAME, subject)  putExtra(NoteIntents.EXTRA\_TEXT, text)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <activity ...>  <intent-filter>  <action android:name="com.google.android.gms.actions.CREATE\_NOTE" />  <category android:name="android.intent.category.DEFAULT" />  <data android:mimeType="\*/\*" />  </intent-filter>  </activity> | </end> |
| <hitle> | Phone Intent | <chare> | 2 | <pext> | --------------Initiate a phone call----------------------  To open the phone app and dial a phone number,  we can use the ACTION\_DIAL action and specify a phone number  using the URI scheme.  When the phone app opens, it displays the phone number but the user  must press the Call button to begin the phone call.  To place a phone call directly, we can use the ACTION\_CALL action  and specify a phone number using the URI scheme.  When the phone app opens, it begins the phone call;  the user does not need to press the Call button.  The ACTION\_CALL action requires  that we add the CALL\_PHONE permission to your manifest file.  <uses-permission android:name="android.permission.CALL\_PHONE" />  1. Action:  ACTION\_DIAL - Opens the dialer or phone app.  ACTION\_CALL - Places a phone call (requires the CALL\_PHONE permission)  2. Data URI Scheme: [tel:<phone-number](tel:%3cphone-number)>, voicemail:<phone-number>  3. MIME Type: None  fun dialPhoneNumber(phoneNumber: String) {  val intent = Intent(Intent.ACTION\_DIAL).apply {  data = Uri.parse("tel:$phoneNumber")  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  } | </end> |
| <hitle> | Search Intent | <chare> | 2 | <pext> | --------Search using a specific app---------  To support search within the context of the app,  we declare an intent filter with the SEARCH\_ACTION action.  <activity android:name=".SearchActivity">  <intent-filter>  <action android:name="com.google.android.gms.actions.SEARCH\_ACTION"/>  <category android:name="android.intent.category.DEFAULT"/>  </intent-filter>  </activity>  ------------Perform a web search--------------------------  To initiate a web search, we use the ACTION\_WEB\_SEARCH action  and specify the search string in the SearchManager.QUERY extra.  1. Action: ACTION\_WEB\_SEARCH  2. Data URI Scheme: None  3. MIME Type: None  4. Extras:  SearchManager.QUERY - The search string.  fun searchWeb(query: String) {  val intent = Intent(Intent.ACTION\_WEB\_SEARCH).apply {  putExtra(SearchManager.QUERY, query)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  } | </end> |
| <hitle> | Settings Intent | <chare> | 2 | <pext> | --------Open a specific section of Settings--------------  To open a screen in the system settings when the app requires  the user to change something, we can use several intent actions.  1. Action  ACTION\_SETTINGS, ACTION\_WIRELESS\_SETTINGS, ACTION\_AIRPLANE\_MODE\_SETTINGS, ACTION\_WIFI\_SETTINGS, ACTION\_APN\_SETTINGS …  2. Data URI Scheme: None  3. MIME Type: None  fun openWifiSettings() {  val intent = Intent(Settings.ACTION\_WIFI\_SETTINGS)  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  } | </end> |
| <hitle> | Text Messaging Intent | <chare> | 2 | <pext> | --------Compose an SMS/MMS message with attachment--------  To initiate an SMS or MMS text message, we can use several actions  Like ACTION\_SENDTO, and specify message details such as  the phone number, subject, and message body using the extra keys.  1. Action: ACTION\_SENDTO or ACTION\_SEND or ACTION\_SEND\_MULTIPLE  2. Data URI Scheme:  sms:<phone\_number>  smsto:<phone\_number>  mms:<phone\_number>  mmsto:<phone\_number>  3. MIME Type:  "text/plain"  "image/\*"  "video/\*"  4. Extras:  "subject" - A string for the message subject (usually for MMS only).  "sms\_body" - A string for the text message.  EXTRA\_STREAM - A Uri pointing to the image or video to attach.  fun composeMmsMessage(message: String, attachment: Uri) {  val intent = Intent(Intent.ACTION\_SENDTO).apply {  type = HTTP.PLAIN\_TEXT\_TYPE  putExtra("sms\_body", message)  putExtra(Intent.EXTRA\_STREAM, attachment)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  If we want to ensure that the intent is handled only  by a text messaging app and not other email or social apps,  then we can use the ACTION\_SENDTO action and include  the "smsto:" data scheme.  fun composeMmsMessage(message: String, attachment: Uri) {  val intent = Intent(Intent.ACTION\_SEND).apply {  data = Uri.parse("smsto:") // This ensures only SMS apps respond  putExtra("sms\_body", message)  putExtra(Intent.EXTRA\_STREAM, attachment)  }  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <activity ...>  <intent-filter>  <action android:name="android.intent.action.SEND" />  <data android:type="text/plain" />  <data android:type="image/\*" />  <category android:name="android.intent.category.DEFAULT" />  </intent-filter>  </activity> | </end> |
| <hitle> | Web Browser Intent | <chare> | 2 | <pext> | ------------Load a web URL--------------------  To open a web page, we can use the ACTION\_VIEW action and specify  the web URL in the intent data.  1. Action: ACTION\_VIEW  2. Data URI Scheme:  http:<URL>  https:<URL>  3. MIME Type  "text/plain"  "text/html"  "application/xhtml+xml"  "application/vnd.wap.xhtml+xml"  fun openWebPage(url: String) {  val webpage: Uri = Uri.parse(url)  val intent = Intent(Intent.ACTION\_VIEW, webpage)  if (intent.resolveActivity(packageManager) != null) {  startActivity(intent)  }  }  <activity ...>  <intent-filter>  <action android:name="android.intent.action.VIEW" />  <!-- Include the host attribute if you want your app to respond  only to URLs with your app's domain. -->  <data android:scheme="http" android:host="www.example.com" />  <category android:name="android.intent.category.DEFAULT" />  <!-- The BROWSABLE category is required to get links from web pages. -->  <category android:name="android.intent.category.BROWSABLE" />  </intent-filter>  </activity> | </end> |
| <hitle> | Verify Intents with Android Debug | <chare> | 2 | <pext> | To verify that the app responds to the intents that we want to support,  we can use the adb tool to fire specific intents:  We need to set up an Android device for development,  or use a virtual device.  We need to install a version of the app that handles the intents  we want to support.  We can fire an intent using adb:  adb shell am start -a <ACTION> -t <MIME\_TYPE> -d <DATA> \  -e <EXTRA\_NAME> <EXTRA\_VALUE> -n <ACTIVITY>  For example:  adb shell am start -a android.intent.action.DIAL \  -d tel:555-5555 -n org.example.MyApp/.MyActivity  If you defined the required intent filters, the app should handle the intent. | </end> |