CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

DIT341 - Lecture 10:
Mobile Development with Android (2)

Gregory Gay
(Some slides by Grischa Liebel)

This Lecture

- Testing
- Profiling
- Processes and Threads
- Services
- Broadcast Receivers
- Content Providers (if we have time)

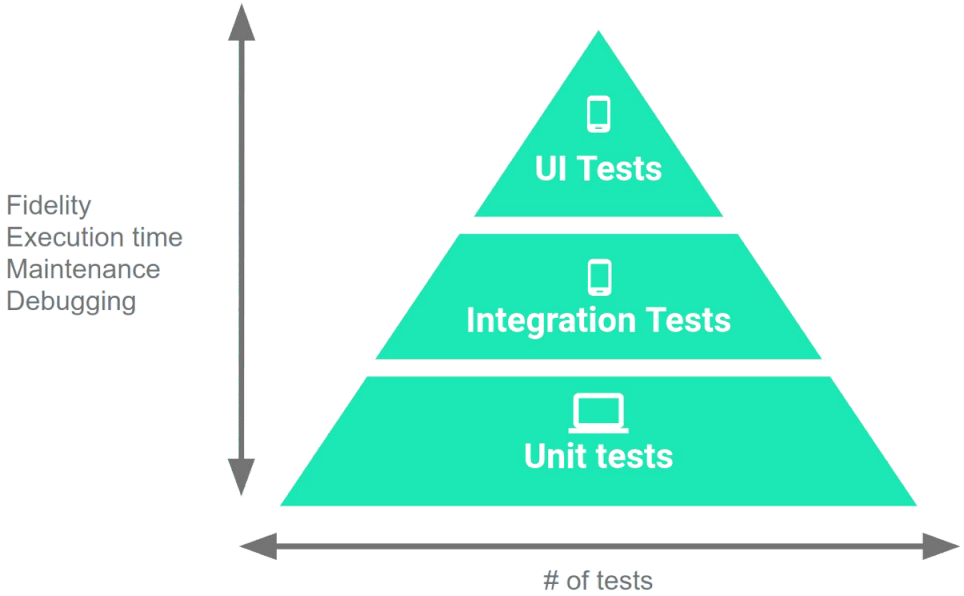
Related Activities

- **Activity 21: Optional**
- Practice Android with Codelabs for Android Developer Fundamentals (V2):
  - https://developer.android.com/courses/fundamentals-training/toc-v2
  - 2.2: Activity lifecycle and state
  - 4.1: Clickable images
  - 4.2: Input controls
  - 4.3: Menus and pickers
  - 4.4: User navigation
  - 4.5: RecyclerView

Testing and Profiling

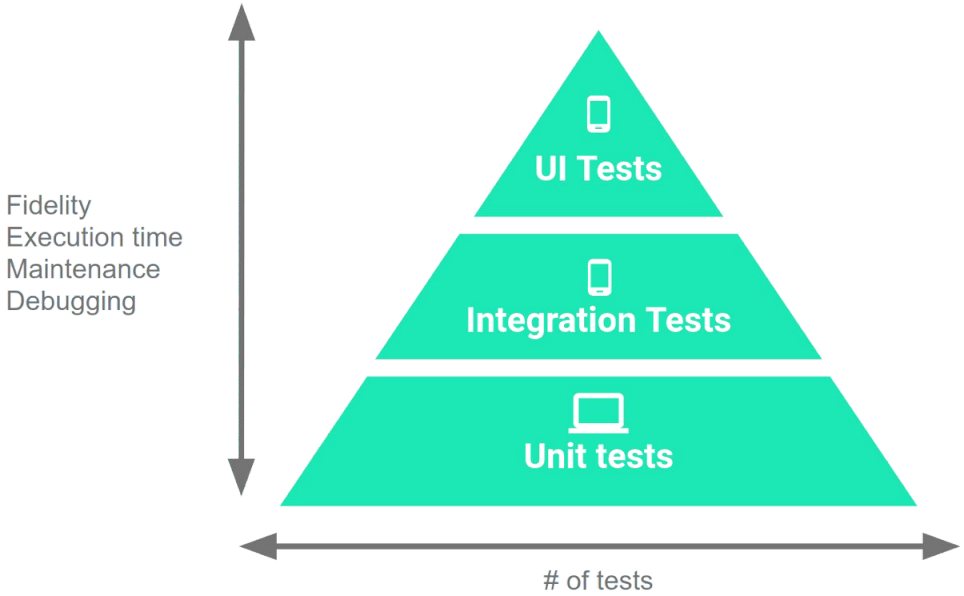
Testing

- Unit tests verify behavior of a single class.
  - 70% of your tests.
- Integration tests verify class interactions in a portion of the app.
  - 20% of your tests.
- UI tests verify end-to-end journey over the app.
  - 10% of your tests.



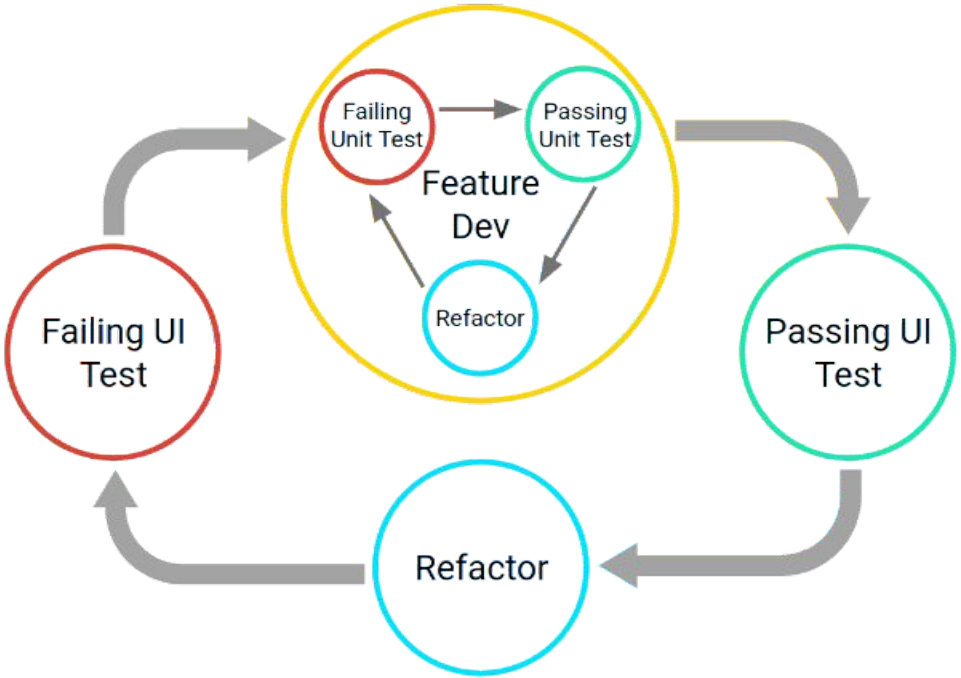
Testing

- 70/20/10 recommended.
- Unit tests execute quickly, without emulator or devices.
- UI tests must run in Android, are very slow.
- Well-tested units reduce likelihood of integration issues, making high levels of testing easier.



Testing Lifecycle

- Create and test code in iterative cycles.
  - Write test cases before you code!
  - It will fail until you finish the feature (correctly).
- Development is a series of nested cycles.
  - Features tested, and integrated (tested via UI)



Thinking in Terms of Modules

- Develop and test app as a set of modules.
  - Clusters of classes centered around a task.
  - Business logic, UI elements, data.
  - Ex: Task List App
    - Creating tasks, viewing stats, attaching photographs
- Allows iterative completion of app, independent testing of features.
- Create consistent APIs to allow modules to interact
- Set well-defined boundaries around modules.

Unit Testing

- Test each class (or smallest unit) in isolation.
- Test standard interactions, invalid inputs, resource availability (network, files, database).
- Tests written in JUnit 4.
  - AndroidX Test Library provides resources, allows tests to run locally.

```
@Test
public void ensureTextViewIsCorrect() throws Exception {
    MainActivity activity = rule.getActivity();
    EditText textView = activity.findViewById(R.id.editText);
    assertEquals( expected: "test",textView.getText().toString());
}
```

Testing in Isolation

- Mock objects (AKA: test doubles) can be used to replace dependencies.
  - Simple replacements that offer static answers to method calls and queries.
- If you've already tested dependencies, use those!
  - If not, use mock objects.
- Mock objects also good for long operations, hard to create configurations.
- Mockito library enables mocking.

UI Testing

- Espresso library enables UI testing through JUnit.
- Get views - `onView(withId(R.id.my_view))`
- Perform actions - `View.perform(click())`
- Check results - `View.check(matches(isDisplayed()))`

```
@Test
public void greeterSaysHello() {
    onView(withId(R.id.name_field)).perform(typeText("Steve"));
    onView(withId(R.id.greet_button)).perform(click());
    onView(withText("Hello Steve!")).check(matches(isDisplayed()));
}
```

Unit Test Example

Uses Espresso testing libraries to interact with Views and Intents. (Part of AndroidX)

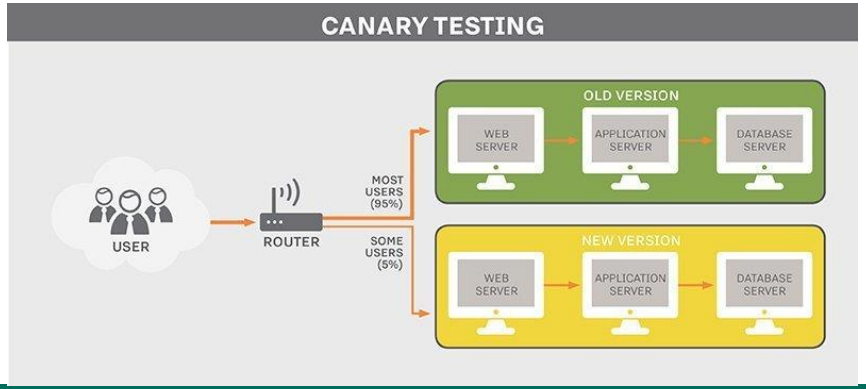
```
@Test
public void successfulLogin() {
    LoginActivity activity =
        ActivityScenario.launch(LoginActivity.class);
    onView(withId(R.id.user_name)).perform(typeText("test_user"));
    onView(withId(R.id.password))
       .perform(typeText("correct_password"));
    onView(withId(R.id.button)).perform(click());
    assertThat(getIntents().first())
        .hasComponentClass(HomeActivity.class);
 }
```

Smoke Testing

- Does it start smoking when you turn it on?
- Extremely basic tests of functionality.
  - Simple starting point for testing.
  - Quick check that all functions work at a basic level.
- Avoid running UI tests every time, just some of the most important at critical points.
  - i.e., login, logout, creating a new account, page loads
  - If smoke tests fail, run deeper tests.

Canary Testing

- Push changes to a small subset of users.
  - Users **do not** volunteer.
- Ensures code changes work in a real environment.
  - If there is a problem, you can revert changes quickly.
  - Only small subset of users impacted.
- Can be automated.



Monkey Testing

- Automatically stress test an app by attacking with random input.
  - UI tests not exhaustive.
  - Monkey good for “filling in some gaps”.
- Tool included in Android SDK.
  - Generates random keystrokes, touches, and gestures.
  - Run through command line.

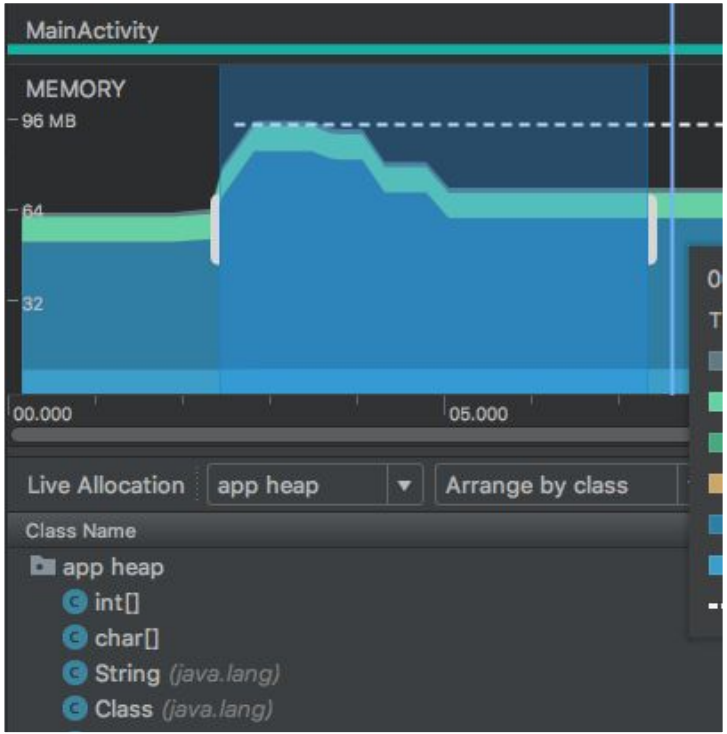
Demonstration - Monkey Testing

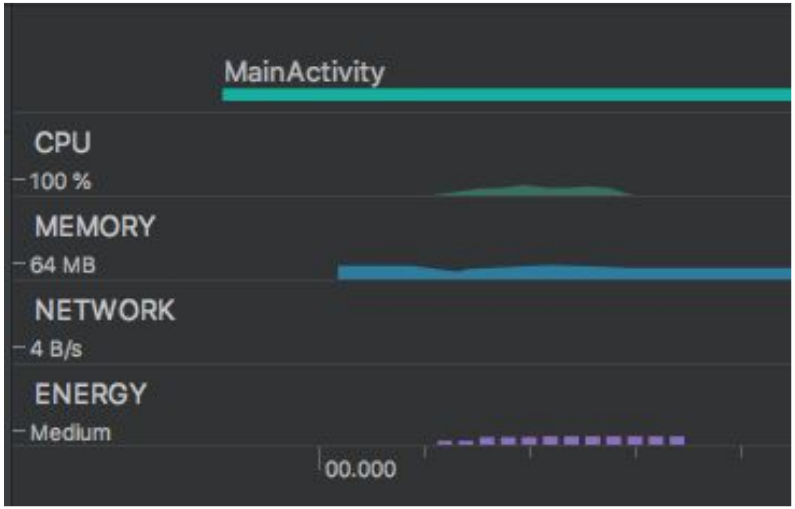
Profiling

- Used to investigate performance of your app.
  - Tool included in Android Studio.

- Find expensive computations.
  - Shows how app uses CPU, memory, network, battery
- Important to understand how to optimize the UX.

Profiling



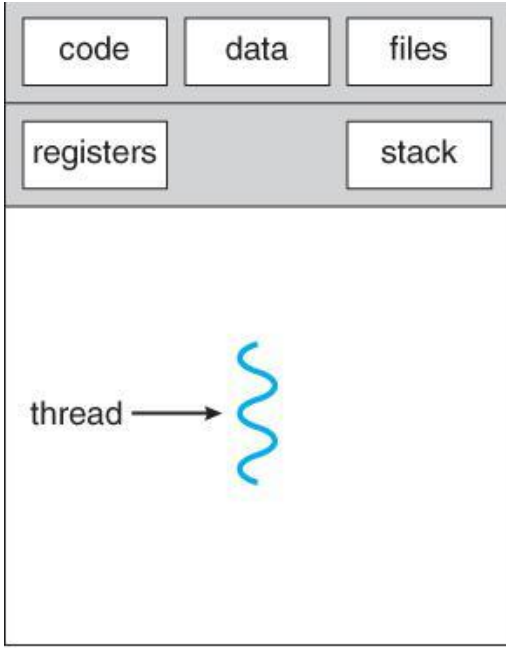


- Detailed views on
  - CPU
  - Memory allocation
  - Network traffic
  - Energy consumption

Backend Processing

Processes and Threads

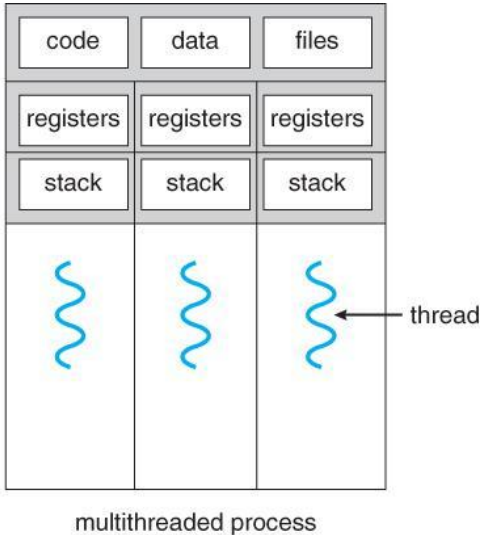
- Independent sequences of execution.
- A running program is a process.
  - Allocated memory, executable code, connections to OS.
  - A process has one or more threads.
  - Each process has a “main/primary” thread.
- Processes are isolated from each other.
  - Communication between processes is expensive.



single-threaded process

Threads

- Entity within a process
  - Can be scheduled for execution
  - Units of execution (“workers”)
- Threads share a memory space.
  - Has own registers and stack.
  - Shares code, data, files.
- Much easier communication, parallel execution.
- But… Concurrency issues.
  - Race conditions, deadlock, ...



multithreaded process

Processes and Threads in Android

- Each app is (by default) started in a single process
- On startup, a main thread is created ("UI thread")
  - Each component is instantiated within this thread
  - Reacts to user events (input)
  - Freezes if you do long-running operations in it!
- Rule: Never block the UI thread!
  - (After 5s, Android asks if you want to close the app)
  - Long-running operations should be done on background threads.
    - Accessing disk, performing network requests.

Specifying Code for a Thread

- Create a class implementing `Runnable`
- Implement `run()`
  - Can't directly modify `View` objects.
  - Set thread priority.
    - Default: background (prevents competition with UI thread)
  - Store reference to the connected thread.

```
class PhotoDecodeRunnable implements Runnable {
...
    /*
     * Defines the code to run for this task.
     */
    @Override
    public void run() {
        // Moves the current Thread into the background
        android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_BACKGROUND);
        ...
        /*
         * Stores the current Thread in the PhotoTask instance,
         * so that the instance
         * can interrupt the Thread.
         */
        photoTask.setImageDecodeThread(Thread.currentThread());
        ...
    }
...
}
```

Communicating with UI Thread

- AFTER performing work, move results of background processing to UI elements.
- Use `Handler` running on UI thread.
  - Handlers receive messages and run associated code.
  - Connect to threads and run code on that thread.
  - Override `handleMessage()`.
    - Invoked when message sent to the managed thread.

```
/*
 * handleMessage() defines the operations to perform when
 * the Handler receives a new Message to process.
 */
@Override
public void handleMessage(Message inputMessage) {
    // Gets the image task from the incoming Message object.
    PhotoTask photoTask = (PhotoTask) inputMessage.obj;
    ...
}
```

Support for Background Processing

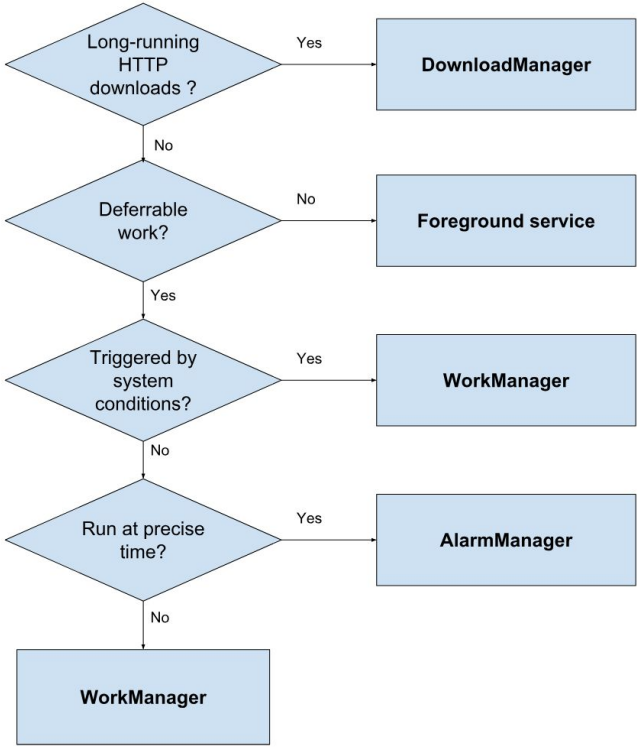
- WorkManager
  - Library that runs background tasks when conditions (network availability, battery) are satisfied.
  - For work that can be put off until later.
- Foreground Services
  - Background process without UI.
  - Work that must run immediately and be completed.
  - Prevents OS from killing process.

Support for Background Processing

- AlarmManager
  - Launches a process at a scheduled time.
  - WorkManager better balances system resources, but AlarmManager is needed for specific timed processes.
    - Use WorkManager for tasks that execute every hour. Use AlarmManager for tasks executing at 6:38 AM.
- DownloadManager
  - Performs background downloading tasks.
  - Provide URI, library handles HTTP interactions, retries, connectivity changes.

How to Handle Background Tasks

- Can the work be deferred, or does it need to happen now?
  - If not, Foreground Service
- Is the work dependent on system conditions?
  - If yes, WorkManager
- Does the job need to run at a precise time?
  - If yes, AlarmManager. If no, WorkManager



Let’s Take a Break

Services

```
<service
    android:name=".MyActualService"
    android:enabled="true"
    android:exported="true" />
```

- Component in the background, no UI
- Often used for long-running operations
  - Play music
  - Process data
  - Send/receive data over the internet
- Will continue to run even if user switches to another app.
- Started with an `Intent`

Services

- Foreground Services
  - Perform an operation that users notice.
    - Play music
  - Must display notification.
  - Keep running even if user isn’t interacting with them.
- Background Services
  - Perform operation not noticeable to user.
    - Compress files

Bound Services

- Service bound to component using `bindService()`
- Offers a client-server interface allowing components to interact with the service.
  - Send requests/receive results
  - Across processes (allowing process communication)
- Multiple components can be bound to one service.
- Service destroyed when all unbind.

Services vs Threads

- Rule of thumb:
  - Operation should run, even if user doesn't interact with the app (or closes it): Service
  - Long-running operation, while the user is active: Thread
- Example: Alarm clock
  - Alarm should go off, even if you close the clock app.
  - Service!
- Example: Buffer video
  - Video should stop if the app is closed.
  - Thread!

Service Basics

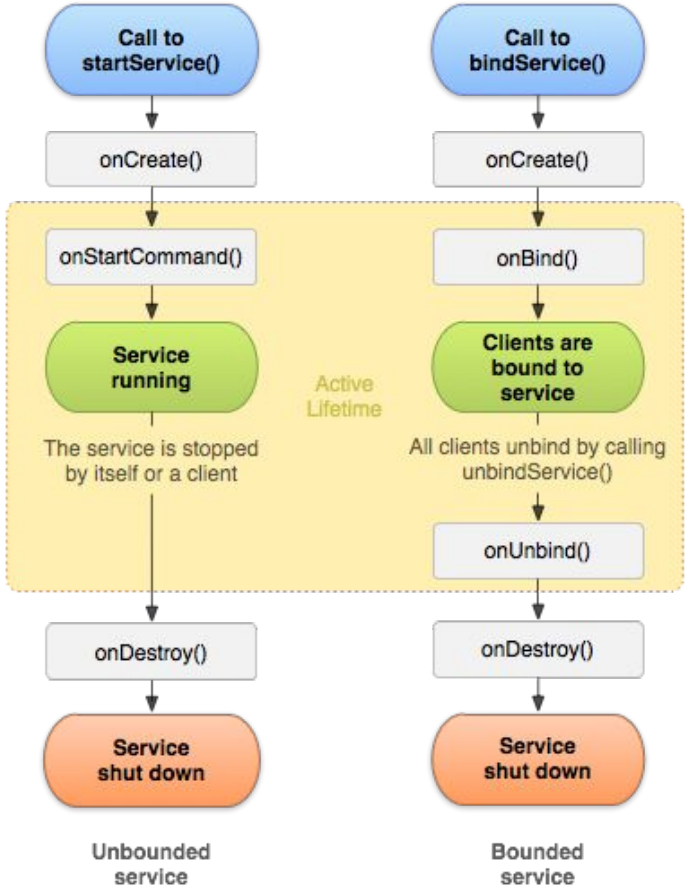
- Create subclass of `Service`
- Implement lifecycle methods:
  - `onStartCommand()`
    - Invoked when another component requests the Service.
    - Service is started and will run until `stopService()` called.
  - `onBind()`
    - Provides interface used to communicate with the Service.
  - `onCreate()`
    - Setup when Service created.
  - `onDestroy()`
    - Cleanup when Service is complete.

Service Basics

- If created by call to `startService()`, Service will run until `stopService()` is called.
- If created by call to `bindService()`, will run only as long as component is bound to it.
  - Once all components unbound, system will kill.
- OS kills Services only if memory is low.
  - Foreground services rarely killed.
  - Services must be designed to gracefully restart if killed before finished with task.

Service Lifecycle

- Lifetime occurs between call to `onCreate()` and return from `onDestroy()`
- Active lifetime begins with call to onStartCommand() or onBind().
  - Intent handed to service and acted on.
- Started services can be bound.



Services vs IntentServices

- Services are multi-threaded and handle requests from multiple components simultaneously.
- Most Services are extensions of `IntentServices`
  - Creates a worker thread that executes all Intents delivered to `onStartCommand()`.
  - Creates a work queue to perform requests.
  - Stops service once all requests are handled.
  - Provides default implementations of `onBind()` and `onStartCommand()`.

IntentServices

- `onHandleIntent()`
  - Performs tasks
  - Returns result
  - Stops the service

```
public class HelloIntentService extends IntentService {

  /**
   * A constructor is required, and must call the super <code><a href="/reference/
   * constructor with a name for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
  }

  /**
   * The IntentService calls this method from the default worker thread with
   * the intent that started the service. When this method returns, IntentService
   * stops the service, as appropriate.
   */
  @Override
  protected void onHandleIntent(Intent intent) {
      // Normally we would do some work here, like download a file.
      // For our sample, we just sleep for 5 seconds.
      try {
          Thread.sleep(5000);
      } catch (InterruptedException e) {
          // Restore interrupt status.
          Thread.currentThread().interrupt();
      }
  }
}
```

Word of Warning

- Services are difficult to get right!
- Much more to cover than fit the slides/time
- If you want to implement a service, read up on what you need to know:

https://developer.android.com/guide/components/services

Broadcast Receivers

- Apps may send or receive **broadcasts** from the OS or other apps.
- Sent when events of interest occur.
  - Phone plugged into power outlet.
- Apps can register to receive specific broadcasts.
  - OS routes broadcasts to registered apps.
- Messaging system across apps.
  - Allows starting and stopping of Services.

Broadcasts

- Sent to all apps registered to receive the event.
- Message wrapped in Intent object.
  - Action string identifies the event.
    - android.intent.action.AIRPLANE_MODE
  - May include additional contextual information.
    - Boolean indicating whether or not airplane mode is on.
- Complete list of broadcast actions in Android SDK
  - BROADCAST_ACTIONS.TXT

Manifest-Declared Receivers

- System launches app when broadcast is sent.
- Specify <receiver> element in manifest.

```
<receiver android:name=".MyBroadcastReceiver"  android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.intent.action.INPUT_METHOD_CHANGED" />
    </intent-filter>
</receiver>
```

- Subclass BroadcastReceiver and implement `onReceive(Context, Intent)`.

```
public class MyBroadcastReceiver extends BroadcastReceiver {
        private static final String TAG = "MyBroadcastReceiver";
        @Override
        public void onReceive(Context context, Intent intent) {
            StringBuilder sb = new StringBuilder();
            sb.append("Action: " + intent.getAction() + "\n");
            sb.append("URI: " + intent.toUri(Intent.URI_INTENT_SCHEME).toString() + "\n");
            String log = sb.toString();
            Log.d(TAG, log);
            Toast.makeText(context, log, Toast.LENGTH_LONG).show();
        }
    }
```

Context-Registered Receivers

- Receive broadcasts as long as the context is valid.
  - Receive broadcasts as long as Activity is not destroyed, or as long as the app is running, or …
- Within the context:
  - Create an instance of `BroadcastReceiver`
  - Create an `IntentFilter` and register the receiver

```
BroadcastReceiver br = new MyBroadcastReceiver();

IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
    filter.addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED);
    this.registerReceiver(br, filter);
```

Sending Broadcasts

- `sendOrderedBroadcast(Intent, String)`
  - Sends broadcasts to one receiver at a time.
  - Each receiver can build on the result or abort the broadcast.
- `sendBroadcast(Intent)`
  - Sends broadcasts to all receivers in undefined order.
  - “Normal” mode.
  - Efficient, but can’t be filtered, aborted, or modified by intermediate receivers.

Sending Broadcasts

- `sendBroadcast(Intent)`

```
Intent intent = new Intent();
intent.setAction("com.example.broadcast.MY_NOTIFICATION");
intent.putExtra("data","Notice me senpai!");
sendBroadcast(intent);
```

- `LocalBroadcastManager.sendBroadcast(Intent)`
  - Sends broadcasts in the same app as sender.
  - Much more efficient, no security issues.

Broadcast Best Practices

- Use local broadcasts as general purpose publisher/subscriber method in app.
- If many apps subscribe to a broadcast in manifest, they will all launch when broadcast is sent.
  - Use context-registered receivers to add restrictions.
  - Android forces the use of context in some cases.
- Do not broadcast sensitive information with implicit intents. Specify permissions, packages, or use local broadcasts.

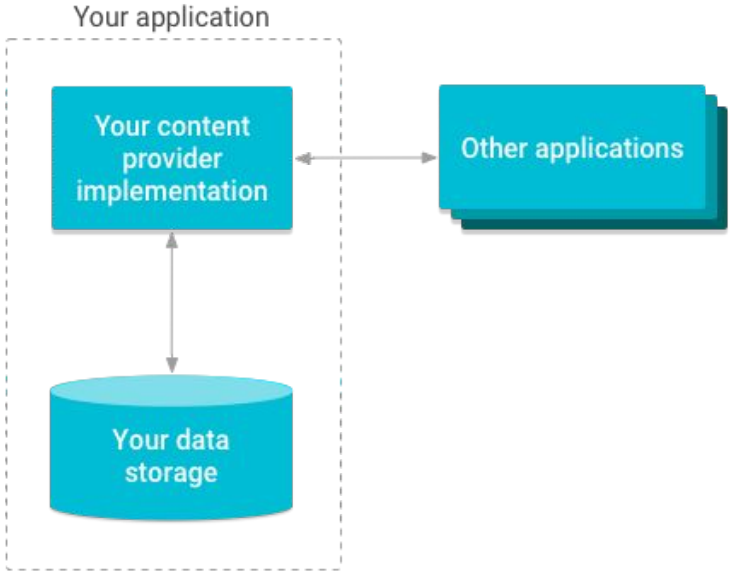
Broadcast Best Practices

- Apps can send malicious broadcasts.
  - Use permissions to block broadcasts, declare "android:exported" false in manifest, use local broadcasts
- Broadcasts use global namespace.
  - Be careful not to conflict when naming.
- `onReceive(..)` runs on main thread.
  - Should execute and return quickly.
- Do not start activities from broadcast receivers.
  - Notifications are more common.

Content Providers

Content Providers

- Provides an interface to a protected data resource.
- Primarily used to share data with other apps in a secure manner.
  - Offer a standard interface for interprocess data sharing.
  - Used when you want to access data stored by another app or share data from your app.

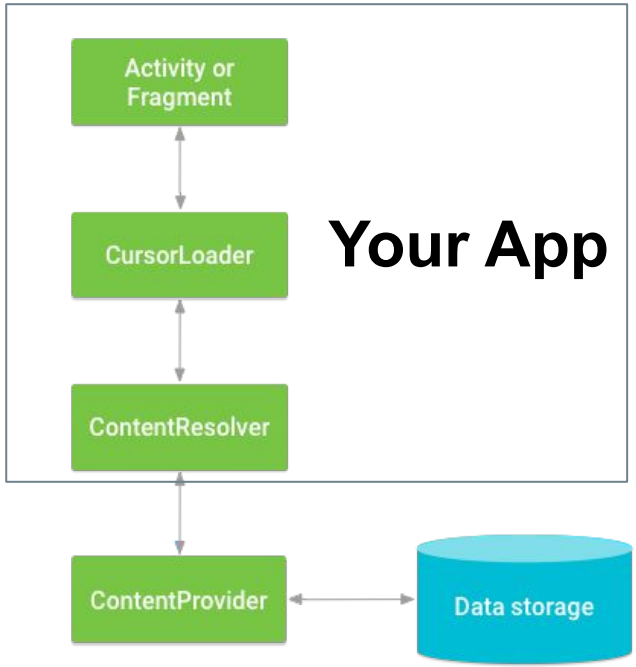


Content Provider Advantages

- Offer control over permissions for accessing data.
  - Can restrict access from within your application, grant access to other applications.
  - Control reading and writing of data.
- Abstracts details of how data is accessed.
  - Data storage type does not matter.
- `CursorLoader` objects offer seamless way to query Content Providers to fill UI elements.

Accessing Content Providers

- You use a `ContentResolver` to communicate with the other app's `ContentProvider`.
  - `ContentProvider` receives data requests and returns results.
- `ContentResolver` provides CRUD functions.
- Commonly, a `CursorLoader` runs a query in the background.
  - Activity calls `CursorLoader`,
  - `CursorLoader` delegates to `ContentResolver`.
  - `ContentResolver` works with `ContentProvider`.
  - This is performed in the background.



Accessing Content Providers

- Android has several built-in Content Providers.
  - One example: user dictionary
  - Rows = words not in a standard dictionary
  - Columns = data attributes associated with the word
    - Includes primary key ID
- To access, call `ContentResolver.query()`.
  - Calls `ContentProvider.query()` defined by the app surfacing the provider.

Access Example

```
// Queries the user dictionary and returns results
cursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
    projection,                         // The columns to return for each row
    selectionClause,                    // Selection criteria
    selectionArgs,                      // Selection criteria
    sortOrder);                         // The sort order for the returned rows
```

Where is the content stored?

Array of columns. What format do you want the result in?

Predicate to filter data. What rows are selected?

Additional optional arguments to filter data (handle missing data, etc.)

How should elements be ordered?

Access Example

```
// A "projection" defines the columns that will be returned for each row
String[] mProjection =
{
    UserDictionary.Words._ID,    // Contract class constant for the _ID column name
    UserDictionary.Words.WORD,   // Contract class constant for the word column name
    UserDictionary.Words.LOCALE  // Contract class constant for the locale column name
};
```

The fields that we want in the result (the projection)

```
/*
 * This defines a one-element String array to contain the selection argument.
 */
String[] selectionArgs = {""};

// Gets a word from the UI
searchString = searchWord.getText().toString();
    // Constructs a selection clause that matches the word that the user entered.
    selectionClause = UserDictionary.Words.WORD + " = ?";

    // Moves the user's input string to the selection arguments.
    selectionArgs[0] = searchString;
```

Does the entry match the first element of the argument array?

Inserting Data

```
// Defines an object to contain the new values to insert
ContentValues newValues = new ContentValues();

/*
 * Sets the values of each column and inserts the word. The arguments to the "put"
 * method are "column name" and "value"
 */
newValues.put(UserDictionary.Words.APP_ID, "example.user");
newValues.put(UserDictionary.Words.LOCALE, "en_US");
newValues.put(UserDictionary.Words.WORD, "insert");
newValues.put(UserDictionary.Words.FREQUENCY, "100");

newUri = getContentResolver().insert(
    UserDictionary.Words.CONTENT_URI,    // the user dictionary content URI
    newValues                           // the values to insert
);
```

Updating Data

Combines updated values (like insertion) with selection criteria (like a query).

```
// Defines an object to contain the updated values
ContentValues updateValues = new ContentValues();

// Defines selection criteria for the rows you want to update
String selectionClause = UserDictionary.Words.LOCALE +  " LIKE ?";
String[] selectionArgs = {"en_%"};

// Defines a variable to contain the number of updated rows
int rowsUpdated = 0;
/*
 * Sets the updated value and updates the selected words.
 */
updateValues.putNull(UserDictionary.Words.LOCALE);

rowsUpdated = getContentResolver().update(
    UserDictionary.Words.CONTENT_URI,    // the user dictionary content URI
    updateValues,                       // the columns to update
    selectionClause,                    // the column to select on
    selectionArgs                       // the value to compare to
);
```

Delete Data

```
// Defines selection criteria for the rows you want to delete
String selectionClause = UserDictionary.Words.APP_ID + " LIKE ?";
String[] selectionArgs = {"user"};

// Defines a variable to contain the number of rows deleted
int rowsDeleted = 0;

// Deletes the words that match the selection criteria
rowsDeleted = getContentResolver().delete(
    UserDictionary.Words.CONTENT_URI,   // the user dictionary content URI
    selectionClause,                    // the column to select on
    selectionArgs                       // the value to compare to
);
```

Not Covered

- **So much more!**
- Creating ContentProviders
- Notifications
- Security
- Packaging/Signing/Publishing Apps
- App Versions
- …
- **More resources: https://developer.android.com/**

What's Next?

- Friday: Project Follow-Up
- Monday: Cloud Deployment
- Milestone 3
  - Android + Presentation

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY