# Data Management

# Different options

- Most Android apps need to save data
  - Shared Preferences for data in key-value pairs
  - Internal Storage on the device memory
  - External Storage on the shared external storage
  - SQLite Databases for structured data in a private database
  - Network connection for data on the web

# Shared preferences

- Class SharedPreferences provides a general framework to save and retrieve persistent key-value pairs of primitive data types (booleans, floats, ints, longs, and strings)
  - These data will persist across user sessions
- To get a SharedPreferences object we can use getSharedPreferences() if we need multiple preferences files, or getPreferences() one preferences file is enough
- Call edit() to get a SharedPreferences.Editor, putBoolean() or putString(), and then commit()
- Call getBoolean() or getString() to read values

```
public class Calc extends Activity {
    public static final String PREFS_NAME = "MyPrefsFile";

    @Override
    protected void onCreate(Bundle state){
       super.onCreate(state);
       . . .

       // Restore preferences
       SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
       boolean silent = settings.getBoolean("silentMode", false);
       setSilent(silent);
    }

    @Override
    protected void onStop(){
       super.onStop();

      // We need an Editor object to make preference changes.
      // All objects are from android.context.Context
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor = settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      // Commit the edits!
      editor.commit();
    }
}
```

# File system

- Internal storage - Always available
  - Files saved here are accessible by only your app
  - System removes all app's files when app un-istalled
  - Best when neither the user nor other apps can access files
- External storage - Not always available
  - Files saved here may be read outside of our control
  - System removes app's files only in particular cases
  - Best place for files that don't require access restrictions
    - Files we want to share with other apps
    - Files the user can access with a computer
  - Permissions in the manifest needed
  - Before you do any work with the external storage you should always check whether the media is available

# SQLite

- Android provides full support for SQLite databases
  - Lightweight database based on SQL
  - Standard SQL syntax
- Any database is accessible by name to any class in the application, but not outside the application
- To create a new database, we must subclass SQLiteOpenHelper and override method onCreate() to execute SQLite commands to create tables

# Key operations

- insert() to add data
- query() to retrieve data, and then use a Cursor to navigate them
- delete() to remove data (rows)
- update() to modify data

# Room Persistence Library



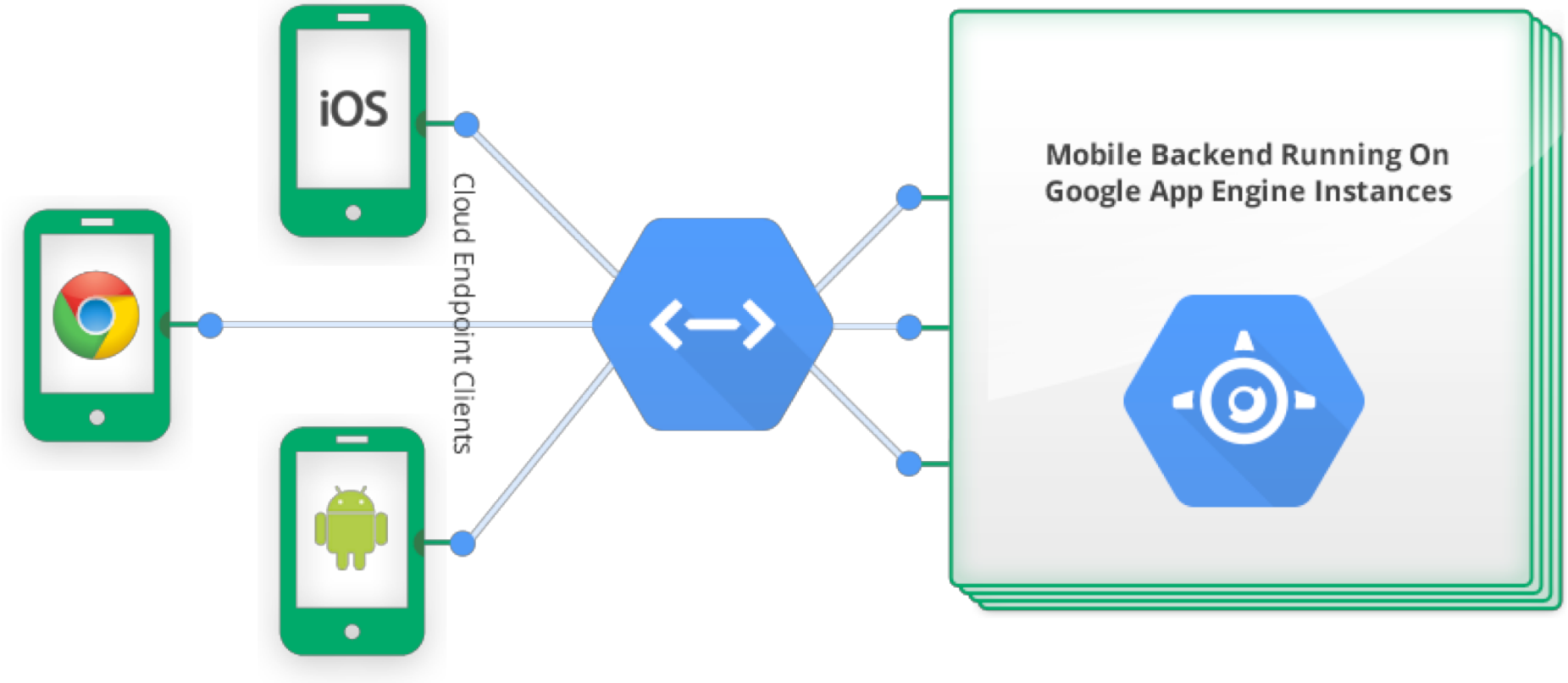
The Room persistence library provides an abstraction layer over SQLite to allow for more robust database access while harnessing the full power of SQLite.

The library helps you create a cache of your app's data on a device that's running your app. This cache, which serves as your app's single source of truth, allows users to view a consistent copy of key information within your app, regardless of whether users have an internet connection.

**Note:** To import Room into your Android project, see adding components to your project.

# Google Cloud Endpoints

- Provides a simple way to develop a shared web backend and also provides critical infrastructures, such as OAuth 2.0 authentication




Firebase

## Develop & test your app

**Realtime Database**
Store and sync app data in milliseconds

**Cloud Firestore**
Store and sync app data at global scale

**Cloud Functions**
Run mobile backend code without managing servers

**Hosting**
Deliver web app assets with speed and security

**Performance Monitoring**
Gain insight into your app's performance

**Crash Reporting**
Find and prioritize bugs; fix them faster

**Authentication**
Authenticate users simply and securely

**Cloud Storage**
Store and serve files at Google scale

**Test Lab for Android**
Test your app on devices hosted by Google

## Grow & engage your audience

**Google Analytics**
Get free and unlimited app analytics

**Dynamic Links**
Drive growth by using deep links with attribution

**Invites**
Make it easy to share your app and content

**AdMob**
Maximize revenue with in-app ads

**Cloud Messaging**
Send targeted messages and notifications

**Remote Config**
Modify your app without deploying a new version

**App Indexing**
Drive search traffic to your mobile app

**AdWords**
Drive installs with targeted ad campaigns

# Retrofit

Download v2.4.0

A type-safe **HTTP client** for Android and Java

## *Introduction*

Retrofit turns your HTTP API into a Java interface.

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}
```

The `Retrofit` class generates an implementation of the `GitHubService` interface.

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();

GitHubService service = retrofit.create(GitHubService.class);
```

Each `Call` from the created `GitHubService` can make a synchronous or asynchronous HTTP request to the remote webserver.

```
Call<List<Repo>> repos = service.listRepos("octocat");
```

Use annotations to describe the HTTP request:

# Volley overview



Volley is an HTTP library that makes networking for Android apps easier and most importantly, faster. Volley is available on GitHub.

Volley offers the following benefits:

- Automatic scheduling of network requests.
- Multiple concurrent network connections.
- Transparent disk and memory response caching with standard HTTP cache coherence.
- Support for request prioritization.
- Cancellation request API. You can cancel a single request, or you can set blocks or scopes of requests to cancel.
- Ease of customization, for example, for retry and backoff.
- Strong ordering that makes it easy to correctly populate your UI with data fetched asynchronously from the network.
- Debugging and tracing tools.

Volley excels at RPC-type operations used to populate a UI, such as fetching a page of search results as structured data. It integrates easily with any protocol and comes out of the box with support for raw strings, images, and JSON. By providing built-in support for the features you need, Volley frees you from writing boilerplate code and allows you to concentrate on the logic that is specific to your app.

# Content providers

- Manage access to a structured set of data
  - Encapsulate data and provide mechanisms for defining data security
  - A content provider presents data to external applications as one or more tables that are similar to the tables found in a relational database
- Are the standard interface that connects data in one process with code running in another process
- Applications usually must request specific permissions in their manifest files to access providers
- There is no need for a provider if you don't intend to share your data with other applications
- Android itself includes content providers that manage data such as audio, video, images, and personal contact information

# Content providers

- Content providers are primarily intended to be used by other applications, which access the provider using a provider client object
- Providers and provider clients offer a consistent, standard interface to data that also handles inter-process communication and secure data access
- An application accesses the data from a content provider with a ContentResolver client object
  - This object provides the basic "CRUD" (create, retrieve, update, and delete) functions of persistent storage
  - ContentResolver has methods that call identically-named methods of ContentProvider
  - ContentProvider also acts as an abstraction layer on top of its repository of data

# mCursor = getContentResolver().query

| query() argument | SELECT keyword/parameter | Notes |
|---|---|---|
| `Uri` | `FROM table_name` | `Uri` maps to the table in the provider named *table_name*. |
| `projection` | `col,col,col,...` | `projection` is an array of columns that should be included for each row retrieved. |
| `selection` | `WHERE col = value` | `selection` specifies the criteria for selecting rows. |
| `selectionArgs` | (No exact equivalent. Selection arguments replace `?` placeholders in the selection clause.) | |
| `sortOrder` | `ORDER BY col,col,...` | `sortOrder` specifies the order in which rows appear in the returned `Cursor`. |

# Threads and services

# How apps work

- The system creates a thread for the application, called “main” or “UI thread”
  - It dispatches events to the user interface
- Everything happens in the UI thread
  - Long operations block the UI
  - No events can be dispatched
- Being blocked for more than 5 secs means “application not responding”

# Threads

- Android natively supports multi-threading
- An application can comprise concurrent threads
- Threads are managed like in Java by
  - Extending class Thread
  - Implementing interface Runnable
    - Method run() is executed by means of method start()

# Some simple rules

- Do not block the UI thread
- Do not access the Android UI toolkit from outside the UI thread
- All manipulations to the UI must be done within the UI thread

# Bad examples

```
public void onClick(View v) {
        Bitmap b = loadImageFromNetwork("http://example.com/image.png");
        myImageView.setImageBitmap(b);
}
```

```
public void onClick(View v) {
    new Thread(new Runnable() {
    public void run() {
        Bitmap b = loadImageFromNetwork( "http://example.com/image.png");
        myImageView.setImageBitmap(b);
        }}).start();
}
```

```
 public void onClick(View v) {
     Bitmap b;
     new Thread(new Runnable() {
     public void run() {
         b = loadImageFromNetwork( "http://example.com/image.png");
     }}).start();
     myImageView.setImageBitmap(b);
 }
```

# How can we fix the problem?

- Different ways to access the UI thread from other threads
  - Activity.runOnUiThread(Runnable)
  - View.post(Runnable)
  - View.postDelayed(Runnable, long)
- The Runnable is sent to the UI thread and runs within it
  - It is invoked on a View from outside the UI thread

# Example

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
        final Bitmap bitmap =
                loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                  mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

# Asynchronous tasks

- Allow one to perform background operations and publish results on the UI thread without manipulating threads
  - One must subclass AsyncTask and implement method doInBackground() that runs in a pool of background threads
- To run the task call execute() from the UI thread
- We can cancel() the task at any time from any thread

# Key elements

- doInBackground() executes automatically on a worker thread
  - This step is used to perform long-running computations in background
  - The result of the computation must be passed back
- onPreExecute(), onPostExecute() and onProgressUpdate() are all invoked on the UI thread
  - The value returned by doInBackground() is sent to onPostExecute()
- We can call publishProgress() at anytime in doInBackground() to execute onProgressUpdate() on the UI thread

# Example

```
public void onClick(View v) {
    new DownloadTask().execute("http://example.com/image.png");
}

private class DownloadTask extends AsyncTask <String, Void, Bitmap> {
    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}
```

# Broadcast receiver

- A component that allows us to register for system or application intents sent by sendBroadcast()
  - All registered receivers for an intent are notified by the Android runtime once the event happens
- A broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work
  - A receiver must be registered
    - Dynamically through registerReceiver()
    - Statically in the manifest through tag <receiver>
- The implementing class extends class BroadcastReceiver
  - Method onReceive() is called by the Android system
    - Once the code returns, the system considers the object to be finished and no longer active

# Events

- Several system events are defined as final static fields in class Intent
- Other Android system classes also define events

| Event | Description |
|---|---|
| Intent.ACTION_BOOT_COMPLETED | Boot completed. Requires the `android.permission.RECEIVE_BOOT_COMPLETED` permission. |
| Intent.ACTION_POWER_CONNECTED | Power got connected to the device. |
| Intent.ACTION_POWER_DISCONNECTED | Power got disconnected to the device. |
| Intent.ACTION_BATTERY_LOW | Triggered on low battery. Typically used to reduce activities in your app which consume power. |
| Intent.ACTION_BATTERY_OKAY | Battery status good again. |

# Intent broadcast mechanism

- Intent objects are delivered to all interested parties
- Android finds the appropriate activity, service, or broadcast receiver to respond to the intent
  - Instantiates them if necessary
- Broadcast intents are delivered only to broadcast receivers, never to activities or services
  - An intent passed to startActivity() is delivered only to an activity, never to a service or broadcast receiver, etc.

# sendBroadcast

- sendBroadcast
  - Broadcasts the given intent to all interested BroadcastReceivers
    - An optional required permission could be enforced
  - This call is asynchronous; it returns immediately
- sendOrderedBroadcast
  - Broadcasts the given intent to all interested BroadcastReceivers
    - Delivering them one at a time to allow preferred receivers to consume the broadcast before it is delivered to the others
  - This call is asynchronous; it returns immediately

# Local broadcasting

- LocalBroadcastManager can be used to register for and broadcast intents to local objects within your application

- Broadcasted data do not leave your app
  - No need to worry about leaking private data
- Other apps cannot communicate with these broadcasts
  - No need to worry about having security holes
- More efficient than sending a global broadcast through the system

# Services

- Application components that perform long-running operations in background without a user interface
  - Must be declared in the manifest
  - Any app component can use a service in the same way as any component can use an activity
    - We can declare the service as private, in the manifest file, and block access from other applications
- Continue to run in background even if the user switches to another application
- A component can also bind to a service to interact with it and perform inter-process communication (IPC)

# Three types

- Scheduled
  - A JobScheduler (API level 21) launches the service
  - The system schedules the jobs for execution at the appropriate times
  - Google recommends that we use JobScheduler to execute background services
- Started
  - An application component (such as an activity) calls startService()
  - It runs in the background indefinitely, even if the component that started it is destroyed
  - Usually, it performs a single operation and does not return a result to the caller. When the operation is complete, it stops itself
- Bound
  - An application component binds to it by calling bindService()
  - Offers a client-server interface that allows components to interact with the service, even across processes with interprocess communication (IPC)
  - It runs only as long as an application component is bound to it
- Services can be started and bound at the same time

# At runtime

- A service runs in the main thread of its hosting process
  - The service does not create its own thread and does not run in a separate process
- If your service is going to do any CPU intensive work you should create a new thread within the service to do that work

# Callback methods (I)

- onStartCommand()
  - The system calls this method when another component requests that the service be started (through startService(Intent))
- onBind()
  - The system invokes this method by calling bindService() when another component wants to bind to the service
  - You must provide an interface that clients use to communicate with the service by returning an IBinder
    - You must always implement this method; however, if you don't want to allow binding, you should return null

# Callback methods (II)

- stopSelf() or stopService(Intent)
  - For the self-termination of the service or for asking for the termination of a service from the outside
  - No need to implement these methods if we only want to provide binding
  - System-decided termination (i.e., memory shortage)
- onCreate()
  - The system calls this method when the service is first created, to perform one-time setup procedures (before it calls either onStartCommand() or onBind())
    - If the service is already running, this method is not called
- onDestroy()
  - The system calls this method when the service is no longer used and is being destroyed

# startService() or bindService()

- If a service is started by invoking startService()
  - It keeps running until it stops itself or another component stops it
- If a service is created by invoking bindService() (and onStartCommand() is not called)
  - It only runs as long as a component is bound to it
  - When the service is unbound from all clients, it is destroyed

# IntentService

- Provides a straightforward solution for handling asynchronous requests (expressed through Intents)
  - onHandleIntent(Intent) must be properly redefined
- Clients send requests through startService(Intent)
  - The service is started as needed and handles Intents using a worker thread automatically
  - The service stops itself as soon it runs out of work
- All requests are handled by a single worker thread
  - Implementation of pattern “Work queue processor”
  - This pattern is commonly used to offload tasks from an application's main thread
  - Only one request will be processed at a time

# Service Notifications

- Once running, a service can notify the user of events using
  - Toast notifications are messages that appear on the surface of the current window for a moment then disappear
  - Status bar notifications provide an icon in the status bar with a message, the user can select it to take an action
    - This is the best technique when some background work has completed

# System services

# Many different services

- Power Service
- KeyGuard Service
- Vibrator Service
- Alarm Service
- Sensor Service
- Audio Service
- Telephony Service
- Connectivity Service
- Wi-Fi Service

# Vibrator Service

```
Vibrator vibrator = (Vibrator)
                getSystemService(Context.VIBRATOR_SERVICE);
```

- Some methods:
  - hasVibrator()
  - vibrate(long time);
  - cancel();
  - vibrate(long[] pattern, int repeat);
- Needs android.permission.VIBRATE

# Sensors

- Android supports three broad categories of sensors
  - Motion sensors: accelerometers, gravity sensors, gyroscopes, and rotational vector sensors
  - Environmental sensors: barometers, photometers, and thermometers
  - Position sensors: orientation sensors and magnetometers
- Sensor framework helps
  - Determine which sensors are available on a device
  - Determine an individual sensor's capabilities
  - Acquire raw sensor data
  - Register and unregister sensor event listeners that monitor sensor changes
- Key elements
  - SensorManager, Sensor, SensorEvent, and SensorEventListener

# Telephony Service

- Provides access to information about the telephony services on the device
  - Applications can use the methods in this class to determine telephony services and states, as well as to access some types of subscriber information
  - Applications can also register a listener to receive notification of telephony state changes
- Some methods
  - getCallState(), getDataState(), getDataActivity(), getNetworkType(), getCellLocation(), getPhoneType(), and isNetworkRoaming()
  - Through usual acquisition of a TelephonyManager

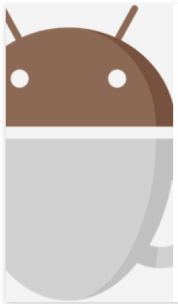
Many other things …

# Testing

# Different types

- Unit testing
- UI testing
- Functional testing
- Integration testing
- Security testing
- Compatibility testing

# Some tools



### Espresso

Use Espresso to write concise, beautiful, and reliable Android UI tests.

### Robolectric

Run unit tests for your app inside the JVM on your workstation.

### AndroidJUnitRunner

Run JUnit 3- or JUnit 4-style test classes on Android devices.

### Android Studio

Use the Android Studio IDE to add and run tests for your app.

# Unit tests

- **Local tests**
  - These tests are compiled to run locally on the Java Virtual Machine (JVM) to minimize execution time
  - Use this approach to run unit tests that have no dependencies on the Android framework or have dependencies that can be filled by using mock objects
- **Instrumented tests**
  - Unit tests that run on an Android device or emulator
  - These tests have access to instrumentation information for the app under test
  - Use this approach to run unit tests that have Android dependencies which cannot be easily filled by using mock objects

# Android-specific extensions to Junit

- MoreAsserts
  - Additional result checking classes
- ViewAsserts
  - Asserts about view layout
- TouchUtils
  - Classes for generating touch events
- Instrumentation
  - For monitoring application interaction with system

# Espresso

- Concise, beautiful, and reliable Android UI tests
  - **Espresso** is the entry point to interactions with views
  - **ViewMatchers** is a collection of objects that implement the Matcher
  - **ViewActions** is a collection of ViewAction objects
  - **ViewAssertions** is a collection of ViewAssertion objects
- Espresso recipes

# ViewActions

- ViewActions.click() clicks on the view
- ViewActions.typeText() clicks on a view and enters a specified string
- ViewActions.scrollTo() scrolls to the view
- ViewActions.pressKey() performs a key press using a specified keycode
- ViewActions.clearText() clears the text in the target view

# Example

```
@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()));
}
```

# Espresso Test Recorder

- Espresso tests consist of two primary components
  - UI interactions include tap and type actions that a person may use to interact with your app
  - Assertions verify the existence or contents of visual elements on the screen
- We can use tests generated by Espresso Test Recorder with Firebase Test Lab to test your app in the cloud on hundreds of device configurations

# UI Automator

- Provides a set of APIs to build UI tests that perform interactions on user apps and system apps
- Is well-suited for writing black box-style automated tests, where the test code does not rely on internal implementation details of the target app
- The key features are:
  - A viewer to inspect layout hierarchy
  - An API to retrieve state information and perform operations on the target device
  - APIs that support cross-app UI testing

# Test Orchestrator

- Minimal shared state
  - Each test runs in isolation (specific instance of Instrumentation)
  - If your tests share app state, most of that shared state is removed from your device's CPU or memory after each test
- Crashes are isolated
  - Even if one test crashes, it takes down only its own instance of Instrumentation
  - The other tests still run

# Monkeyrunner

- Provides an API for writing programs that control an Android device or emulator from outside of Android code
- We can write a Python program that installs an Android application or test package, runs it, sends keystrokes to it, takes screenshots of its user interface, and stores screenshots on the workstation
- One can develop an entire system of Python-based modules and programs for controlling Android devices

# UI/Application Exerciser Monkey

- Is a program that runs on an emulator or device and generates pseudo-random streams of user events
- It includes a number of options:
  - Basic configuration options, such as setting the number of events to attempt
  - Operational constraints, such as restricting the test to a single package
  - Event types and frequencies
  - Debugging options

# Other testing tools

- Robotium
- Uiautomator
- Robolectric
- Calabash
- Appium
- …