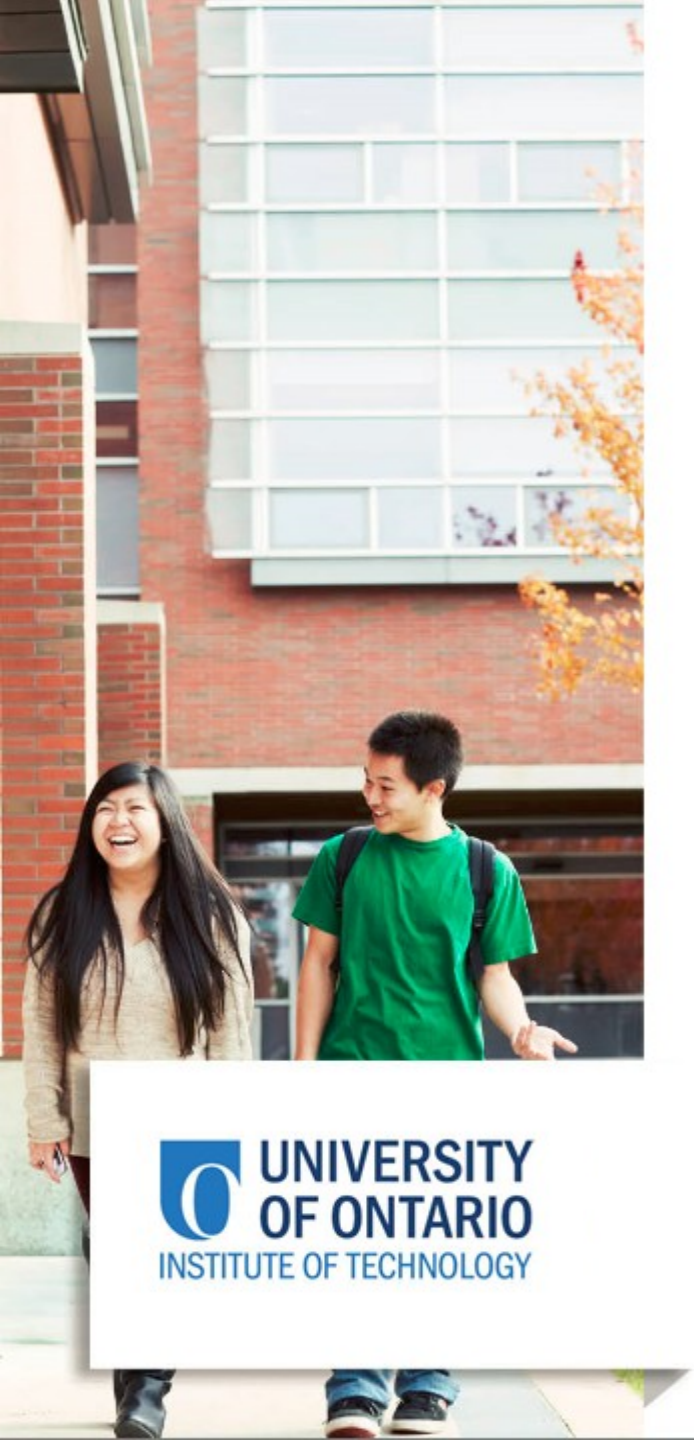


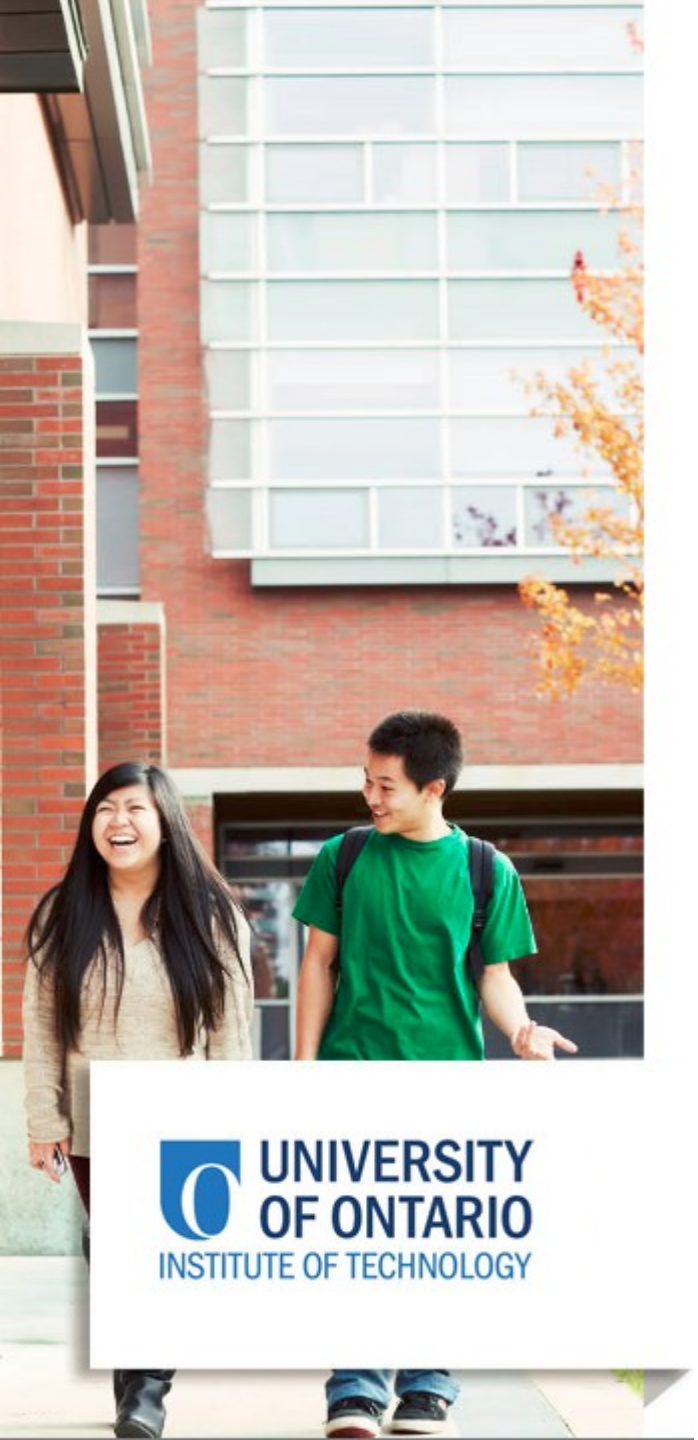
Mobile Devices

Internet Resources and Saving State

Randy J. Fortier
Randy.Fortier@uoit.ca
@randy_fortier

Outline

- In this section, we'll learn how to:
  - Download and use Internet resources
    - Files from the Internet
  - Avoid a seemingly-frozen user interface by doing major processing in the background
  - Multi-threading
  - AsyncTask
  - How to save an application's state in Android
    - Using SharedPreferences
    - Using lifecycle methods
    - Using private (and public) files
    - Cloud storage
    - Database operations – Discussed later



Android

Internet Resources

Internet Resources

- This could include:
  - Embedding a WebView into your app
  - Downloading an XML from a web site
    - e.g. Weather updates, RSS, directions
  - Interacting with a well-known web service
    - e.g. Google API, PayPal, Amazon WS

Downloading Arbitrary Files

```
URL url = URL("http://www.abc.com/image.png");
HttpURLConnection conn;
conn = (HttpURLConnection)url.openConnection();
int result = conn.getResponseCode();
if (result == HttpURLConnection.HTTP_OK) {
  InputStream in = conn.getInputStream();
  // read the PNG data
}
```

Granting Permission

- This code won't work unless your application has permission to access the Internet:

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

Example XML Feeds

- Sports scores:
`http://www.scorespro.com/rss/live-soccer.xml`
- Stocks:
`http://www.nasdaq.com/aspxcontent/NasdaqRSS.aspx?data=uvup`
- Exchange rates:
`http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml`
- Weather:
`http://www.nhc.noaa.gov/index-at.xml`

Example XML Document (RSS)

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Soccer Livescore by ScoresPro.com</title>
    <description>Soccer Livescore results in real time by
                 ScoresPro.com</description>
    <pubDate>Tue, 2 Oct 2015 17:44:49 GMT</pubDate>
    <item>
      <title>Soccer Livescore: (POL-FA) #Zaglebie Lubin vs #Polonia
             Warszawa: 1-1</title>
      <description>2nd Half Started</description>
      <pubDate>Tue, 2 Oct 2015 17:39:14 GMT</pubDate>
      <link>http://www.scorespro.com/</link>
    </item>
    ...
  </channel>
</rss>
```

Parsing XML (Using DOM)

```
File xmlFile = new File("rss.xml");
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder = dbFactory.newDocumentBuilder();
Document document = docBuilder.parse(xmlFile);
document.getDocumentElement().normalize();

NodeList itemNodes = document.getElementsByTagName("item");

for (int i = 0; i < itemNodes.getLength(); i++) {
   Node itemNode = itemNodes.item(i);
   if (itemNode.getNodeType() == Node.ELEMENT_NODE) {
      Element itemElement = (Element)itemNode;
      String title = getTagValue("title", itemElement);
      ...
   }
}
```

- This code goes through an XML (RSS) document and collects the title of each item
  - You can see how easy it is to parse XML with DOM

Downloading in Activities

- Downloading a parsing an XML file from the Internet is computationally expensive
  - This sort of thing should not happen in the main thread of the application
  - The alternatives:
    - Run the code in a separate thread
    - Use an AsyncTask

Threads

- A thread is similar to a process
  - Only more lightweight to create/manage
- If an application has only one thread, it can't do certain things
  - e.g. If you want to download a file, the user interface would suddenly become unusable
  - e.g. Click on a dropdown list, nothing happens

UI Thread

- In Android, the main thread of each activity is called its UI thread
  - This thread takes care of event handling, animations related to GUI components, etc.
    - These are behaviours associated with responsiveness
  - Imagine if you selected a dropdown, but it took 5 seconds to drop down?
    - You can't do much work in this thread

Multi-threading

- If you have multiple threads, the application becomes more complicated
  - You can consider activities to have their own threads
  - Inactive activities, however, also have idle threads

Running in Another Thread

- Let's say we want to call some method in our activity
  - `doSomethingExpensive();`

```
Runnable runnable = new Runnable() {
  public void run() {
    doSomethingExpensive();
  }
};
Thread thread = new Thread(runnable);
```

AsyncTask

- AsyncTask is a specialized class for performing heavy computation outside of the UI thread
- Implementer:
  - doInBackground(...)
  - onPostExecute()
  - onProgressUpdate()
- Caller:
  - task.execute()

AsyncTask

- The AsyncTask class takes three type parameters

1. The type of arguments passed to the main method
   - doInBackground()
2. The type of a progress metric (e.g. % as int)
3. The type of the result passed back from doInBackground()

AsyncTask

- AsyncTask life cycle:
  - doInBackground(...arguments...)
    - Argument(s) and return type depend on type parameters specified when created
    - Performs the majority of the processing
  - onPostExecute(result)
    - Cleans up after execution
      - e.g. Handles any errors that occurred
  - onProgressUpdate(metric)
    - Called when the progress should be updated
    - Used by activities to display a progress bar

AsyncTask

```
public class FeedUpdaterTask
        extends AsyncTask<String, Void, String> {
  private Exception exception = null;

  @Override
  protected String doInBackground(String... params) {
    // do something long here
    return "I did all my work";
  }

  @Override
  protected void onPostExecute(String result) {
    if (exception != null) {
      exception.printStackTrace();
    }
  }
}
```

AsyncTask

```
public class FeedUpdaterTask
        extends AsyncTask<String, Void, String> {
  private Exception exception = null;

  @Override
  protected String doInBackground(String... params) {
    // do something long here
    return "I did all my work";
  }

  @Override
  protected void onPostExecute(String result) {
    if (exception != null) {
      exception.printStackTrace();
    }
  }
}
```

Parameter Type

AsyncTask

```
public class FeedUpdaterTask
        extends AsyncTask<String, Void, String> {
  private Exception exception = null;

  @Override
  protected String doInBackground(String... params) {
    // do something long here
    return "I did all my work";
  }

  @Override
  protected void onPostExecute(String result) {
    if (exception != null) {
      exception.printStackTrace();
    }
  }
}
```

Progress Type

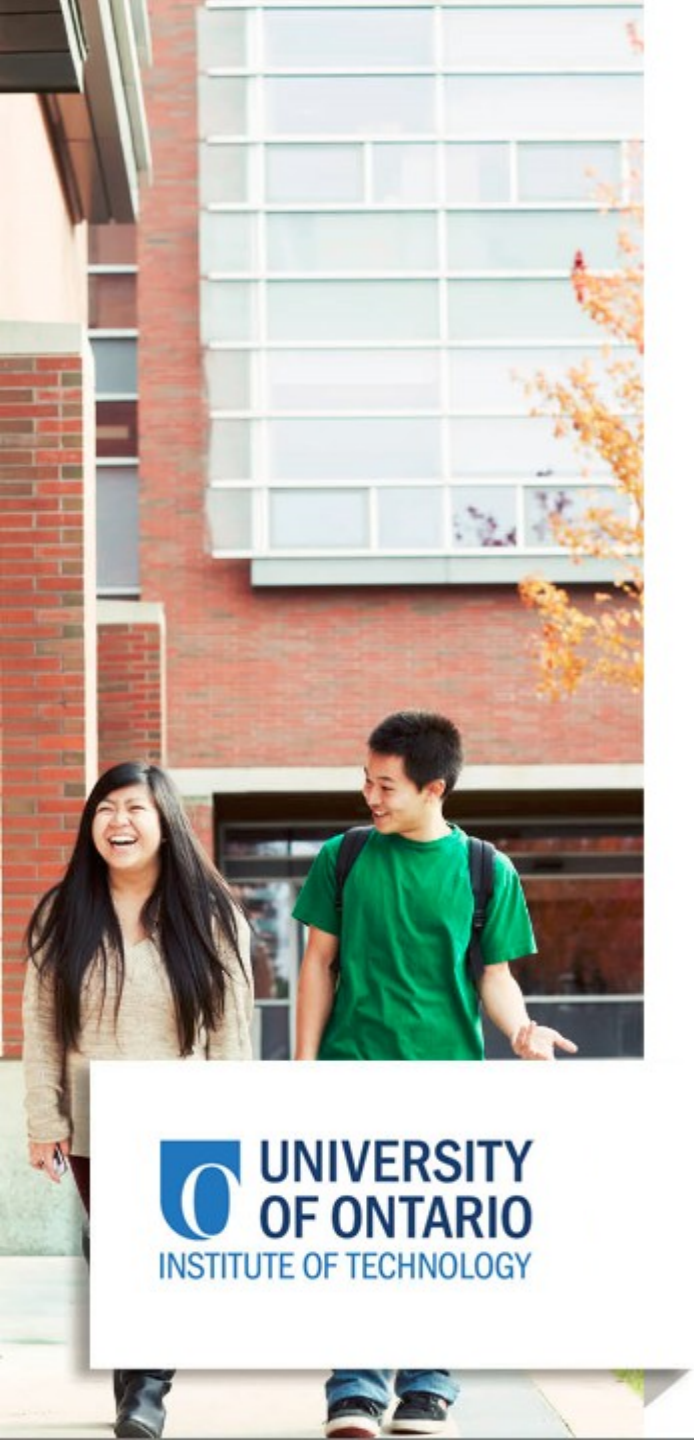
AsyncTask

```
public class FeedUpdaterTask
        extends AsyncTask<String, Void, String> {
  private Exception exception = null;

  @Override
  protected String doInBackground(String... params) {
    // do something long here
    return "I did all my work";
  }

  @Override
  protected void onPostExecute(String result) {
    if (exception != null) {
      exception.printStackTrace();
    }
  }
}
```

Result Type

AsyncTask

```
public class FeedUpdaterTask
             extends AsyncTask<String, Integer, String> {
  @Override
  protected String doInBackground(String... params) {
    // do something long here
    for (int i = 0; i < 100000; i++) {
       if ((i%1000) == 0)
         publishProgress(i / 1000); // updates progress
    }
    return "I did all my work";
  }

  @Override
  protected void onProgressUpdate(Integer... progress) {
    setProgressPercent(progress[0]);
  }
}
```



Android

Saving State – Accessing Files and Preferences

Saving State

- Applications need to save their state
  - For user convenience when they reopen your application
  - For good practice, since your activities may be unloaded from memory completely in memory runs low

Saving State

- Android provides five primary methods:
  - Save your state to the shared preferences object
  - Save your state when the lifecycle method fires
  - Save your state to a file on the mobile device’s file system
  - Save your state to the database
  - Save your state to the cloud

Saving State

- Android provides five primary methods:
  - Save your state to the shared preferences object
    - Data is stored as name/value pairs (like a hash table)
    - As the name indicates, this method is often used to store options/preferences (non-mission-critical)
  - Save your state when the lifecycle method fires
  - Save your state to a file on the mobile device's file system
  - Save your state to the database
  - Save your state to the cloud

Saving State

- Android provides five primary methods:
  - Save your state to the shared preferences object
  - Save your state when the lifecycle method fires
    - Data is stored as name/value pairs (like a hash table)
    - This method is very simple, but should only be used to store small amounts of non-mission-critical data
  - Save your state to a file on the mobile device's file system
  - Save your state to the database
  - Save your state to the cloud

Saving State

- Android provides five primary methods:
  - Save your state to the shared preferences object
  - Save your state when the lifecycle method fires
  - Save your state to a file on the mobile device's file system
    - There isn't really anything special about files in Android versus any other Java system (java.io)
      - The files are normally stored on the device's internal SD card storage
    - You can store data in binary files or text files
    - You can use this to store any type of non-mission-critical data, but managing the data format is up to you
  - Save your state to the database
  - Save your state to the cloud

Saving State

- Android provides five primary methods:
  - Save your state to the shared preferences object
  - Save your state when the lifecycle method fires
  - Save your state to a file on the mobile device's file system
  - Save your state to the database
    - Android supports SQLite databases out-of-the-box
      - Other databases can be installed, as well
    - This is the primary mechanism used to store the actual data of the application (e.g. contact list, high scores)
      - Can even be used for mission-critical data
  - Save your state to the cloud

Saving State

- Android provides five primary methods:
  - Save your state to the shared preferences object
  - Save your state when the lifecycle method fires
  - Save your state to a file on the mobile device's file system
  - Save your state to the database
  - Save your state to the cloud
    - Traditionally, this could be using Internet resources:
      - However, instead of merely connecting to a web site, we could send data (e.g. parameters) along with the request
      - Data could be downloaded using XML, JSON, YAML, HTML, …
      - Details of doing this will not be discussed in this course
    - With Android 4.4+, you can use the Storage Application Framework (SAF):
      - Any cloud storage vendor that supports it will then be accessible
      - e.g. your own proprietary service, Google Drive, iCloud, Amazon Cloud Drive, Amazon S3, Microsoft SkyDrive, Ubuntu One
      - SAF works using Intents

Shared Preferences

- Shared preferences are a lightweight way to store a little information
  - Preferences/options
  - User interface state (selected checkboxes, typed text)
- The shared preferences name/value pairs are serialized to a file by Android

Shared Preferences

```
String name = "myAppPrefs"; // file name
SharedPreferences prefs =
              getSharedPreferences(name, Activity.MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();
editor.putString("lastName", lastName);
editor.putString("lastName", lastName);
editor.putLong("startDate", startDateEpoch);
// no file operations have occurred to this point
editor.apply(); // change asynchronously

SharedPreferences prefs =
              getSharedPreferences(name, Activity.MODE_PRIVATE);
String lastName = prefs.getString("lastName", null);
String firstName = prefs.getString("firstName", null);
long startDateEpoch = prefs.getLong("startDate", null);
```

Shared Preferences

```
String name = "myAppPrefs";
SharedPreferences prefs =
              getSharedPreferences(name, Activity.MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();
editor.putString("lastName", lastName);
editor.putString("lastName", lastName);
editor.putLong("startDate", startDateEpoch);
boolean result = editor.commit(); // apply synchronously

SharedPreferences prefs =
              getSharedPreferences(name, Activity.MODE_PRIVATE);
String lastName = prefs.getString("lastName", null);
String firstName = prefs.getString("firstName", null);
long startDateEpoch = prefs.getLong("startDate", null);
```

Shared Preferences

- By the name of the class, you might guess that preferences can be shared
- Shared Preferences privacy modes:
  - MODE_PRIVATE
    - Accessed only by the application that created it
  - MODE_WORLD_READABLE
    - Read by any application with read access to the file
  - MODE_WORLD_WRITEABLE
    - Written by any application with write access to the file
  - MODE_MULTI_PROCESS
    - Can be written to by other processes, so the file should be re-examined when reading values

PreferenceActivity

- A subclass of Activity used for preferences-related controllers
  - This makes it easier to specifically create activities that are preferences-related
- Also, there are specialized views and layouts for preferences
  - So that preferences have the same look and feel as other preferences screens on Android

Lifecycle Methods

- onSaveInstanceState(Bundle)
  - Called when the activity is being unloaded
  - Here, you would put your state into the bundle
- onCreate(Bundle)
  - Called when the activity is re-created
  - Here, you would restore the state from the bundle

Lifecycle Methods

```
@Override
public void saveInstanceState(Bundle state) {
   EditText fName = (EditText)findViewById(R.id.fname);
   EditText lName = (EditText)findViewById(R.id.lname);
   EditText startDate = (EditText)findViewById(R.id.startDate);
   state.putString("FirstName", fName.getText());
   state.putString("LastName", fName.getText());
   state.putLong("StartDate", Long.valueOf(longstartDate.getText()));
   super.onSaveInstanceState(state);
}

@Override
public void onCreate(Bundle state) {
   super.onCreate(state);
   EditText fName = (EditText)findViewById(R.id.fname);
   EditText lName = (EditText)findViewById(R.id.lname);
   EditText startDate = (EditText)findViewById(R.id.startDate);
   fName.setText(state.getString("FirstName"));
   lName.setText(state.getString("LastName"));
   startDate.setText("" + state.getLong("StartDate"));
}
```

Local Files

- You can also store your data into files on the SD card of the mobile device
  - Text can be stored in a text file
  - Other data can be stored in a binary file
    - e.g. Java objects, compressed data
- Just like StoredPreferences files, these files can be:
  - MODE_PRIVATE
  - MODE_WORLD_READABLE
  - MODE_WORLD_WRITEABLE

Local Files

- Reading from resource files:

```
InputStream raw =
       getResources().openRawResource(R.raw.someFile);
BufferedReader in = new BufferedReader(raw);
String line = in.readLine();
while (line != null) {
  // do something with line
  line = in.readLine();
}
inRaw.close();
```

Local Files

- Reading from files:

```
FileInputStream raw = openFileInput("file.txt");
BufferedReader in = new BufferedReader(raw);
String line = in.readLine();
while (line != null) {
  // do something with line
  line = in.readLine();
}
inRaw.close();
```

Local Files

- Writing to private files:

```
FileOutputStream raw = openFileOutput("file.txt",
                                      Context.MODE_PRIVATE);
PrintWriter out = new PrintWriter(raw);
out.println("<my-xml-data>");
out.println("  <customer name='Bob Smith'>");
…
out.println("</my-xml-data>");
raw.close();
```

Local Files

- Writing to public files:

```
FileOutputStream raw = openFileOutput("file.txt",

                                Context.MODE_WORLD_READABLE);
PrintWriter out = new PrintWriter(raw);
out.println("<my-xml-data>");
out.println("  <customer name='Bob Smith'>");
…
out.println("</my-xml-data>");
raw.close();
```

Local Files

- Listing available private files:

```
String[] files = getContext().fileList();
for (int i = 0; i < files.length; i++) {
   Log.i("MyApp", "File: " + files[i]);
}
```

Local Files

- Deleting a private file:

```
getContext().deleteFile("file.txt");
```

Cloud Files

- Storage access framework
  - This allows access to cloud files, along with shared files on the device itself
- Picking a file:

```
final int OPEN_IMAGE_REQUEST = 41000002;

Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT);
intent.addCategory(Intent.CATEGORY_OPENABLE);
   // only openable files
intent.setType("image/*"); // only images
startActivityForResult(intent, OPEN_IMAGE_REQUEST);
```

Cloud Files

- Reading the data from the file:

```
@Override
public void onActivityResult(int requestCode,
                             int resultCode,
                             Intent resultData) {
    if (requestCode == OPEN_IMAGE_REQUEST && resultCode == RESULT_OK) {
        if (resultData != null) {
            Uri uri = resultData.getData();
            try {
                fd = getContentResolver().openFileDescriptor(uri, "r");
            } catch (FileNotFoundException e) {
                e.printStackTrace();
                return;
            }
            FileInputStream in =
                           FileInputStream(fd.getFileDescriptor());
            ...
        }
    }
}
```

Local Files

- Any functionality provided by java.io is also available
  - e.g. java.io.File
    - File::exists()
    - File::isDirectory()
    - File::mkdir(), File::mkdirs()
    - File::renameTo(File)
    - File::setLastModified(long)
    - File::setReadOnly()
    - File::File::toURL()

Wrap-Up

- In this section, we learned:
  - Download files from the Internet
  - Run code in another thread
  - Use the AsyncTask class to run code outside of the UI thread
  - How to store and retrieve state for an Android application
    - Using SharedPreferences
    - Using lifecycle methods
    - Using private (and public) files
  - We also introduced two other techniques:
    - Cloud storage