Keishin Yokomaku (@KeithYokoma) - DevFest Tokyo 2017

Retro/Prospective Android Application Development

About Me

- 横幕圭真 - Keishin Yokomaku

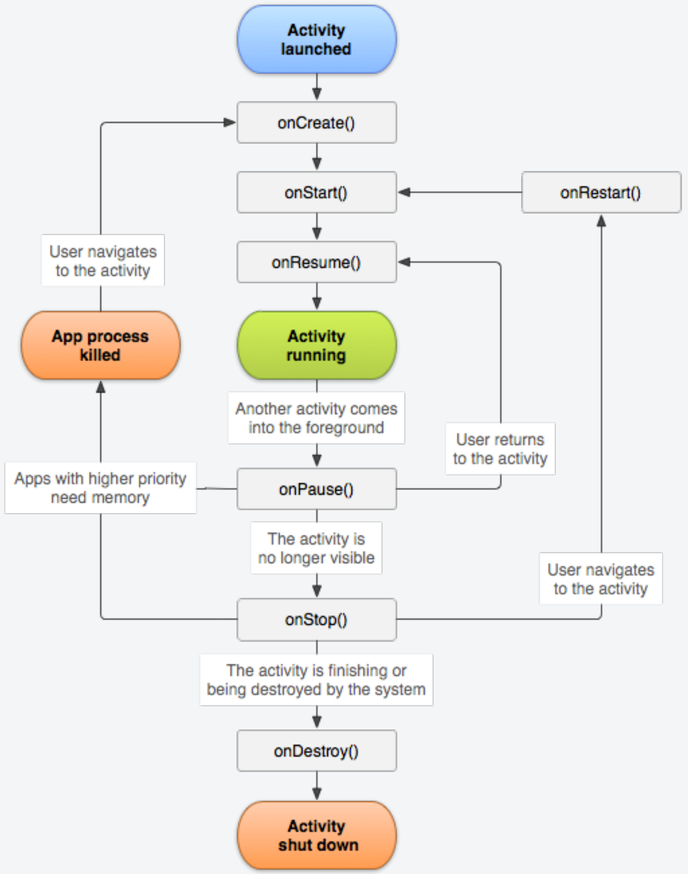
Drivemode, Inc. / Principal Engineer

KeithYokoma: GitHub / Twitter / Qiita / Tumblr / Stack Overflow

- Books: TechBooster / Development guide for mobile app
- Fun: Gymnastics / Cycling / Photography / Motorsport / Hiking

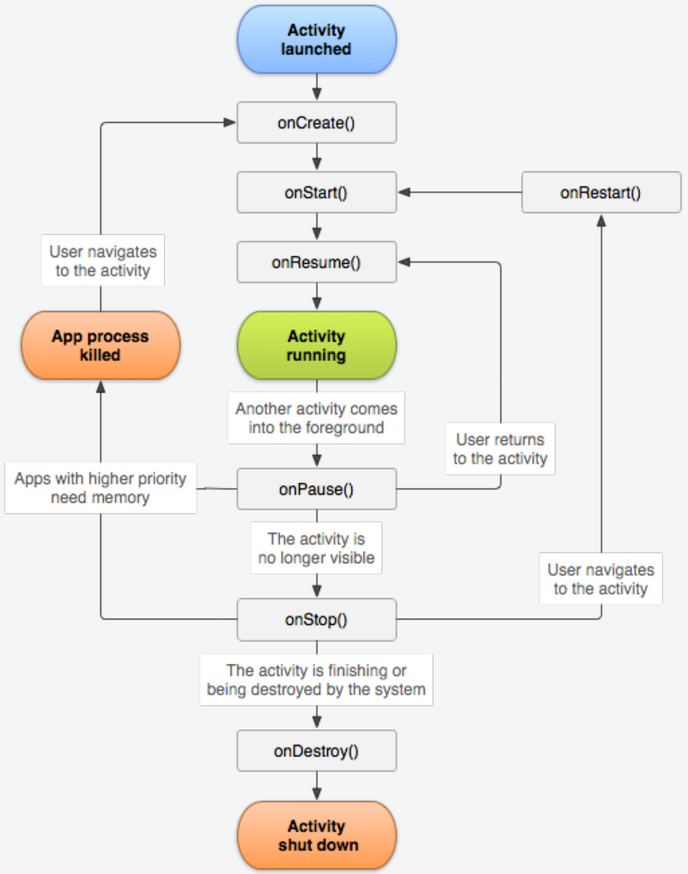
Retro/Prospective Android Application Development

Do you know of...



Back in time to Android 1.5

Activity Lifecycle



Activity Lifecycle

画面を最初に立ち上げた時

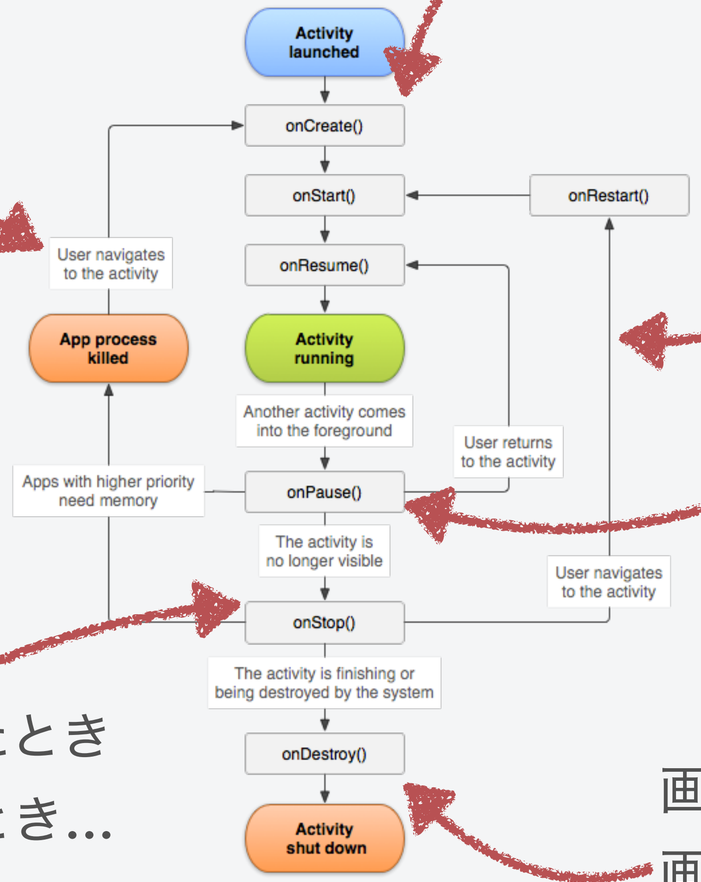
別のアプリから戻ってきて
自分のプロセスが死んでいたとき

別の画面から戻ってきたとき

ダイアログを表示したとき
画面の一部が後ろに見えているとき

ホームボタンをおしたとき
別の画面へ遷移したとき...

画面を閉じたとき
画面を回転したとき...



Activity Lifecycle

- Activity は絶えず状態が遷移する
  - 画面遷移による状態の遷移
  - 端末の状態変化（回転や設定変更など）による状態の遷移
  - 他のアプリの状態変化（メモリの使用状況など）による状態の遷移
  - etc...

Activity Lifecycle

- 少ないメモリ空間を効率的に使うための仕組み
  - ユーザに見えているものが最も重要
  - ユーザが直近で使ったものはすぐ再開できるようになっている
  - 使わなくなりそうなものからどんどんメモリから追い出す
  - メモリがたくさん必要なアプリを使うときは直近で使ったものも追い出す

Activity Lifecycle

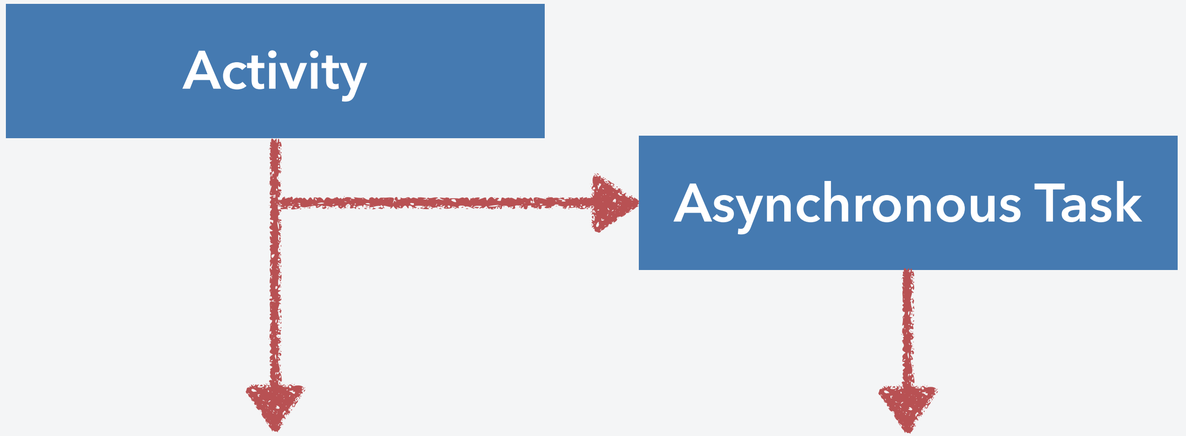
- 端末の設定や状態(Context)の変化に追従しやすくする仕組み
  - 画面が回転したらレイアウトを作り直す
  - 言語設定を変えたら言語リソースを読み直す
  - 夜になったら画像リソースを読み直す

Activity Lifecycle

- アプリに固有な状態を管理するためのメソッド
  - Activity#onCreate(Bundle)
  - Activity#onRestoreInstanceState(Bundle)
  - Activity#onSaveInstanceState(Bundle)
  - Activity#onRetainNonConfigurationInstance()
  - Activity#getLastNonConfigurationInstance()

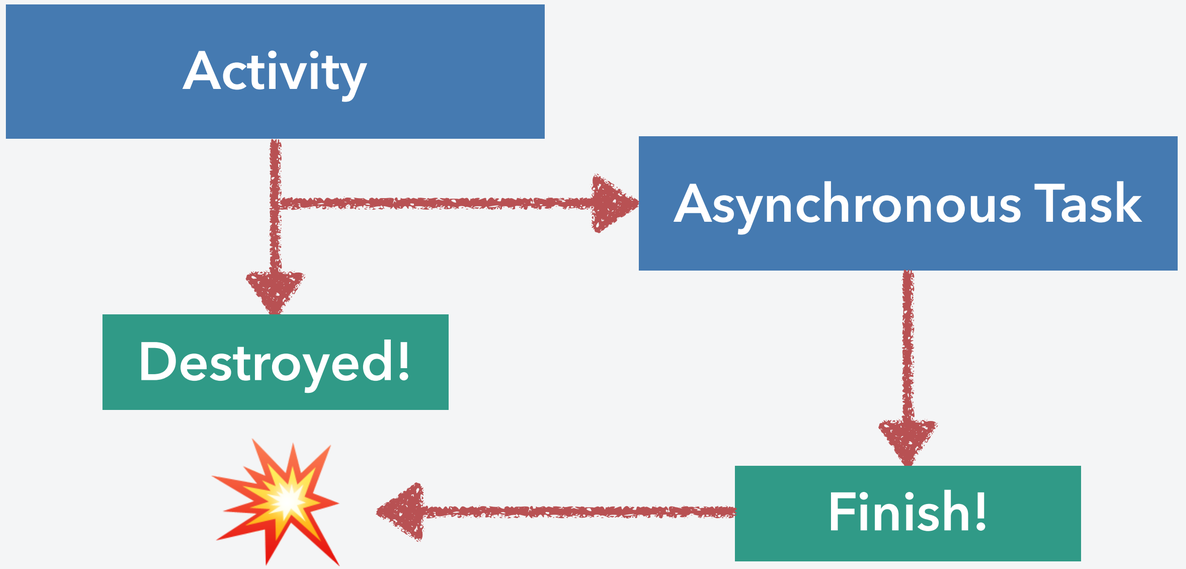
Activity Lifecycle and Asynchronous Task Lifecycle

- Http通信など時間のかかる処理は別スレッドで実行する(非同期処理)
  - UI スレッドで実行するとその処理が終わるまで UI が操作不能になる



Activity Lifecycle and Asynchronous Task Lifecycle

- UI スレッドで動く Activity と別スレッドで動く非同期処理は別のライフサイクル
  - Activity が生きている間に非同期処理が終わるとは限らない



非同期処理 - Before AsyncTask

- スレッドでの処理と UI へのコールバックは全て自分で書く💪
  - スレッドプールなども自前で準備
  - コールバックのために Handler をつかう

非同期処理 - AsyncTask

- 非同期処理のライフサイクルを表す抽象クラス
  - AsyncTask#onPreExecute(): 非同期処理の実行前のコールバック
  - AsyncTask#doInBackground(): 非同期処理の中身
  - AsyncTask#onPostExecute(): 非同期処理の実行後のコールバック
- スレッドの管理は AsyncTask がやってくれる

非同期処理 - AsyncTask

```
public class MainActivity extends Activity {
  static class MyAsyncTask extends AsyncTask<Void, Void, Void> {
    private WeakReference<MyActivity> mRef;

    public MyAsyncTask(MyActivity activity) {
      mRef = new WeakReference<>(activity);
    }

    public Void doInBackground(Void… args) {
      // do something heavy
      return null;
    }

    public void onPostExecute(Void result) {
      // callback to activity
    }
  }
}
```

非同期処理 - AsyncTask

```
public class MainActivity extends Activity {
  static class MyAsyncTask extends AsyncTask<Void, Void, Void> {
    private WeakReference<MyActivity> mRef;

    public MyAsyncTask(MyActivity activity) {
      mRef = new WeakReference<>(activity);
    }

    public Void doInBackground(Void… args) {
      // do something heavy
      return null;
    }

    public void onPostExecute(Void result) {
      // callback to activity
    }
  }
}
```

非同期処理 - AsyncTask

```
public class MainActivity extends Activity {
  static class MyAsyncTask extends AsyncTask<Void, Void, Void> {
    private WeakReference<MyActivity> mRef;

    public MyAsyncTask(MyActivity activity) {
      mRef = new WeakReference<>(activity);
    }

    public Void doInBackground(Void… args) {
      // do something heavy
      return null;
    }

    public void onPostExecute(Void result) {
      // callback to activity
    }
  }
}
```

非同期処理 - AsyncTask

```
public class MainActivity extends Activity {
  static class MyAsyncTask extends AsyncTask<Void, Void, Void> {
    private WeakReference<MyActivity> mRef;

    public MyAsyncTask(MyActivity activity) {
      mRef = new WeakReference<>(activity);
    }

    public Void doInBackground(Void… args) {
      // do something heavy
      return null;
    }

    public void onPostExecute(Void result) {
      // callback to activity
    }
  }
}
```

非同期処理 - AsyncTask

```
public class MainActivity extends Activity {
  static class MyAsyncTask extends AsyncTask<Void, Void, Void> {
    private WeakReference<MyActivity> mRef;

    public MyAsyncTask(MyActivity activity) {
      mRef = new WeakReference<>(activity);
    }

    public Void doInBackground(Void… args) {
      // do something heavy
      return null;
    }

    public void onPostExecute(Void result) {
      // callback to activity
    }
  }
}
```

非同期処理 - AsyncTask

```
public class MainActivity extends Activity {
  private MyAsyncTask mTask;

  public void onButtonClick(View view) {
    if (mTask != null)
      return;
    mTask = new MyAsyncTask(this);
    mTask.execute();
  }

  public void onFinishAsyncTask() {
    mTask = null;
  }
}
```

非同期処理 - AsyncTask

```
public class MainActivity extends Activity {
  private MyAsyncTask mTask;

  public void onButtonClick(View view) {
    if (mTask != null)
      return;
    mTask = new MyAsyncTask(this);
    mTask.execute();
  }

  public void onFinishAsyncTask() {
    mTask = null;
  }
}
```

非同期処理 - AsyncTask

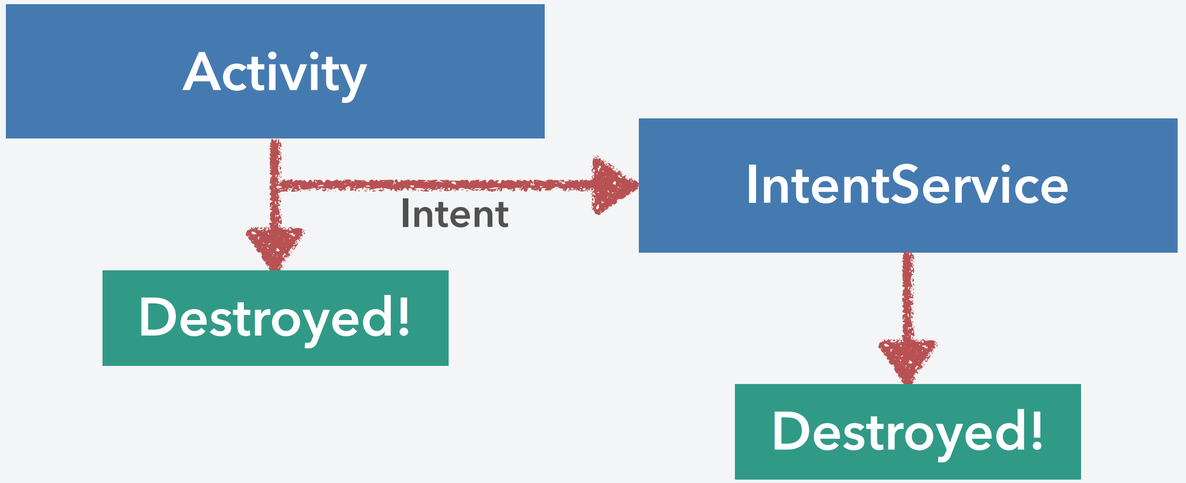
```
public class MainActivity extends Activity {
  private MyAsyncTask mTask;

  public void onButtonClick(View view) {
    if (mTask != null)
      return;
    mTask = new MyAsyncTask(this);
    mTask.execute();
  }

  public void onFinishAsyncTask() {
    mTask = null;
  }
}
```

AsyncTask Issues

- Activity の実装と密結合になりやすかった
  - interface で解決
- コールバックを弱参照で保持しないとメモリリークした
  - Activity と非同期処理のライフサイクルの差異は埋められていない
- 長時間かかる計算処理には向かない

もうひとつの非同期処理 - IntentService

- UI とは関連のない、バックグラウンド処理のためのクラス
- 簡単なジョブキューとして使える
  - ただしメモリ上にしかジョブがないので揮発する



非同期処理 - IntentService

```
public class MainIntentService extends IntentServie {
  public MainIntentService() { super(“MainIntentService”); }
  public MainIntentService(String name) { super(name); }

  @Override
  protected void onHandleIntent(Intent intent) {
    // 非同期処理の中身
  }
}
```

非同期処理 - IntentService

```
public class MainIntentService extends IntentServie {
  public MainIntentService() { super(“MainIntentService”); }
  public MainIntentService(String name) { super(name); }

  @Override
  protected void onHandleIntent(Intent intent) {
    // 非同期処理の中身

  }
}
```

Network I/O

- Apache HttpClient
  - Apache が出している汎用の Http 通信ライブラリ
  - やりたいことは大体できる
  - 巨大
  - Android の API に組み込まれている

Network I/O

- Http(s)URLConnection
  - Java の API に組み込まれている Http 通信ライブラリ
  - シンプルで軽い
  - Android の API に組み込まれている
  - Froyo(Andorid 2.2)までバグっていた

Disk I/O

- ファイルに書き出す各種 API
  - File API (java.io)
  - SQLite
  - SharedPreferences

Disk I/O

- SharedPreferences に書き込むメソッド
  - commit(): 初期からある。ファイルに同期書き込み。
  - apply(): Gingerbread から。メモリに書いたものを後で非同期書き込み。

パフォーマンスのモニタリング - StrictMode

- メインスレッドをブロックする処理の検知
  - ネットワーク通信
  - ディスク I/O
- リソースの閉じ忘れも検知できる
  - Closeable, Cursor など

パフォーマンスのモニタリング - StrictMode

```
public class MyApplication extends Application {
  public void onCreate() {
    if (BuildConfig.DEBUG) {
      StrictMode.enableDefaults();
    }
  }
}
```

パフォーマンスのモニタリング - StrictMode

```
public class MyApplication extends Application {
  public void onCreate() {
    if (BuildConfig.DEBUG) {
      StrictMode.enableDefaults();
    }
  }
}
```

パフォーマンスのモニタリング - StrictMode

```
public class MyApplication extends Application {
  public void onCreate() {
    if (BuildConfig.DEBUG) {
      StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
        .detectDiskReads()
        .detectDiskWrites()
        .detectNetwork()
        .penaltyLog()
        .build());
      StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
        .detectLeakedSqlLiteObjects()
        .detectLeakedClosableObjects()
        .penaltyDeath()
        .build());
    }
  }
}
```

パフォーマンスのモニタリング - StrictMode

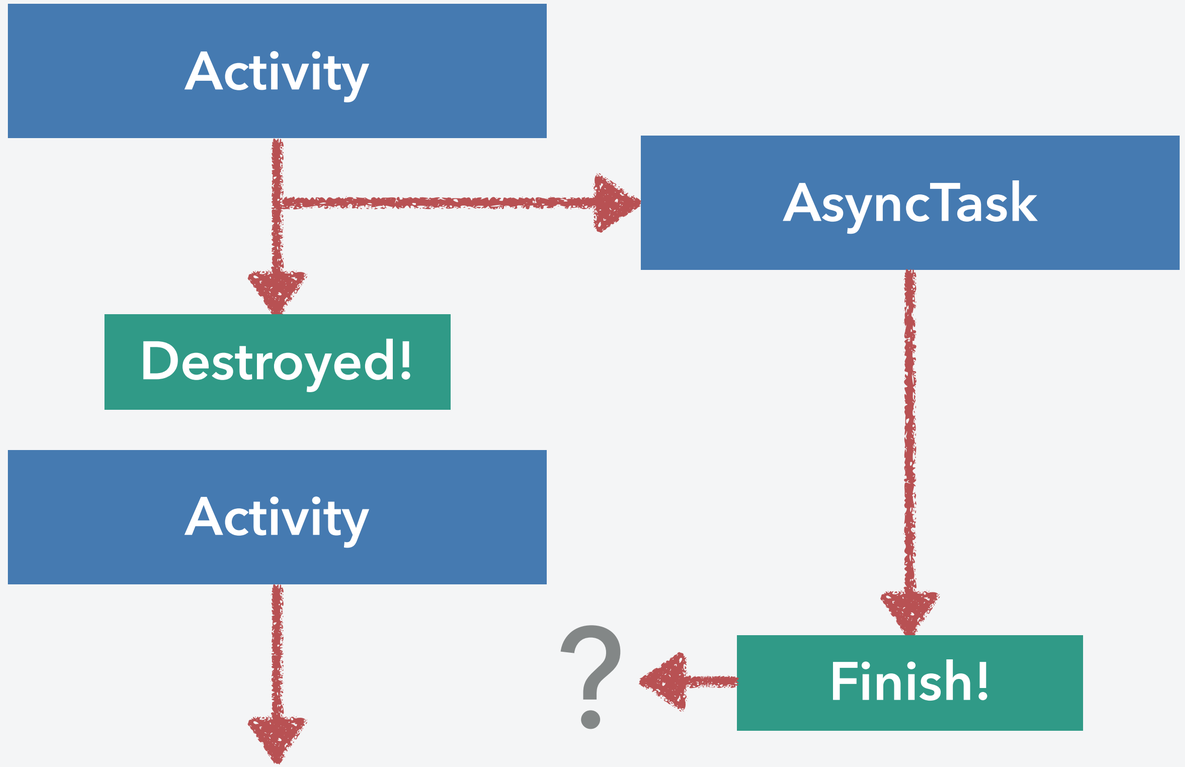
```
public class MyApplication extends Application {
  public void onCreate() {
    if (BuildConfig.DEBUG) {
      StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
        .detectDiskReads()
        .detectDiskWrites()
        .detectNetwork()
        .penaltyLog()
        .build());
      StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
        .detectLeakedSqlLiteObjects()
        .detectLeakedClosableObjects()
        .penaltyDeath()
        .build());
    }
  }
}
```

パフォーマンスのモニタリング - StrictMode

```
public class MyApplication extends Application {
  public void onCreate() {
    if (BuildConfig.DEBUG) {
      StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
        .detectDiskReads()
        .detectDiskWrites()
        .detectNetwork()
        .penaltyLog()
        .build());
      StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
        .detectLeakedSqlLiteObjects()
        .detectLeakedClosableObjects()
        .penaltyDeath()
        .build());
    }
  }
}
```

AsyncTask Issues

- Activity が再生成されたとき

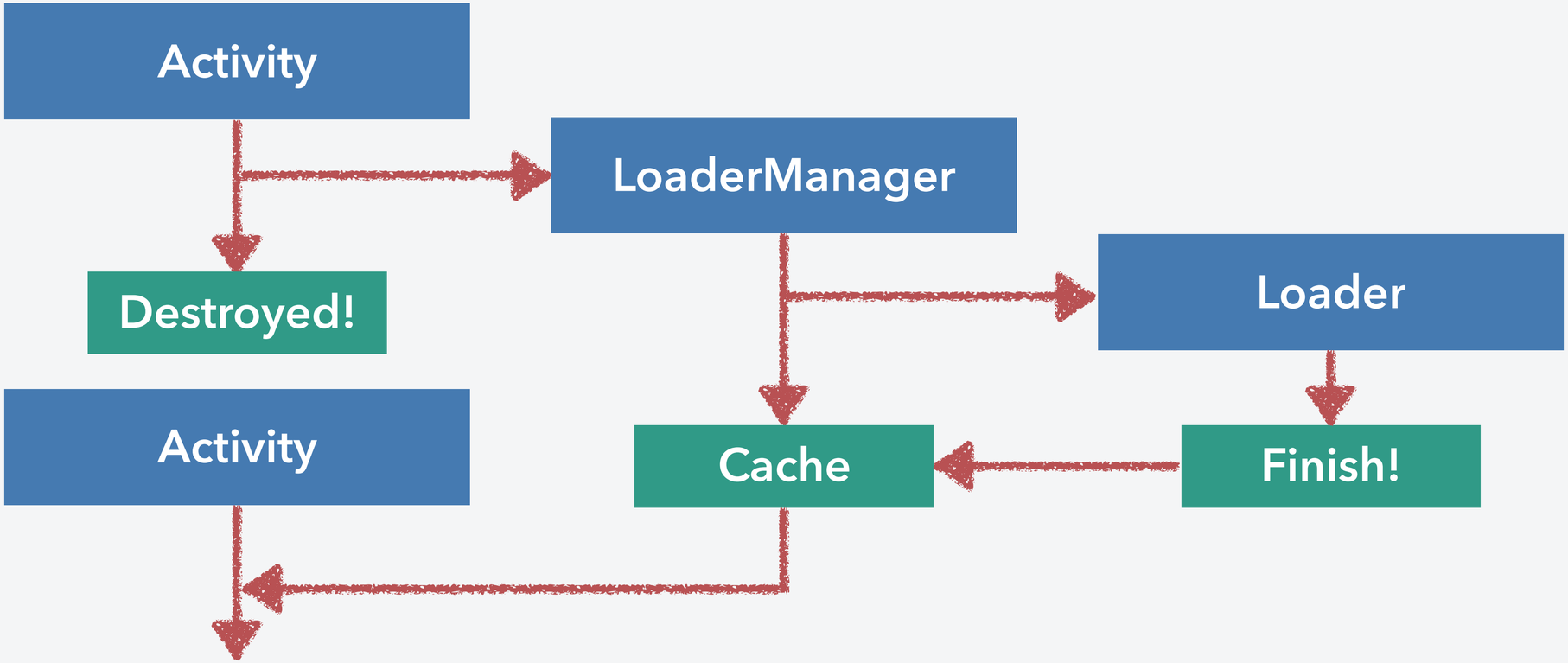


AsyncTask Issues

- Activity が再生成されたとき
  - AsyncTask のインスタンスを再生成後に引き継いで、コールバックを設定しなおす
  - 一度キャンセルして再生成後にやりなおす

Loader

- Activity のライフサイクルと非同期処理のライフサイクルの差異をハンドルする



Loader

```
public class MainActivity extends Activity implements LoaderCallbacks<Data> {
   public Loader<Data> onCreateLoader(int id, Bundle args) {
     if (id == 0) {
       return new MainLoader();
     } else if (id == 1) {
       return new SecondLoader();
     } else {
       throw new IllegalArgumentException(“invalid id”);
     }
   }

   public void onLoadFinished(Loader<Data> loader, Data data) {
     showData(data);
   }

   public void onLoaderReset(Loader<Data> loader) {
   }
}
```

Loader

```
public class MainActivity extends Activity implements LoaderCallbacks<Data> {
   public Loader<Data> onCreateLoader(int id, Bundle args) {
     if (id == 0) {
       return new MainLoader();
     } else if (id == 1) {
       return new SecondLoader();
     } else {
       throw new IllegalArgumentException("invalid id");
     }
   }

   public void onLoadFinished(Loader<Data> loader, Data data) {
     showData(data);
   }

   public void onLoaderReset(Loader<Data> loader) {
   }
}
```

Loader

```
public class MainActivity extends Activity implements LoaderCallbacks<Data> {
   public Loader<Data> onCreateLoader(int id, Bundle args) {
     if (id == 0) {
       return new MainLoader();
     } else if (id == 1) {
       return new SecondLoader();
     } else {
       throw new IllegalArgumentException("invalid id");
     }
   }

   public void onLoadFinished(Loader<Data> loader, Data data) {
     showData(data);
   }

   public void onLoaderReset(Loader<Data> loader) {
   }
}
```

Loader

```
public class MainActivity extends Activity implements LoaderCallbacks<Data> {
   public Loader<Data> onCreateLoader(int id, Bundle args) {
     if (id == 0) {
       return new MainLoader();
     } else if (id == 1) {
       return new SecondLoader();
     } else {
       throw new IllegalArgumentException("invalid id");
     }
   }

   public void onLoadFinished(Loader<Data> loader, Data data) {
     showData(data);
   }

   public void onLoaderReset(Loader<Data> loader) {
   }
}
```

Loader

```
public class MainActivity extends Activity implements LoaderCallbacks<Data> {
   public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_main);

     LoaderManager loaderManager = getLoaderManager();
     loaderManager.initLoader(0, new Bundle(), this);
   }
}
```

Loader

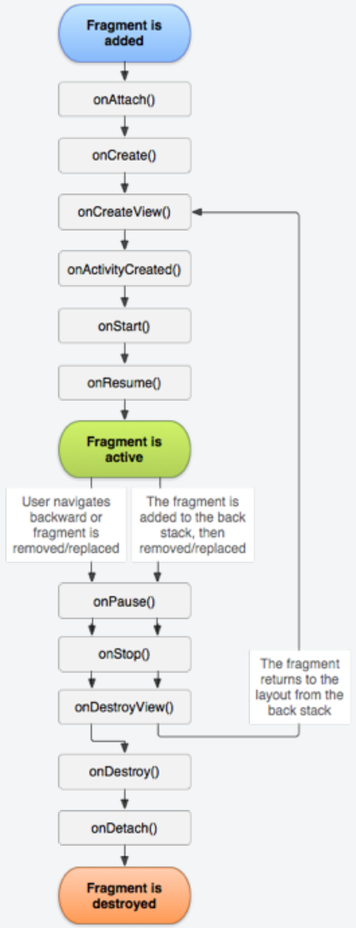
```
public class MainActivity extends Activity implements LoaderCallbacks<Data> {
   public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_main);

     LoaderManager loaderManager = getLoaderManager();
     loaderManager.initLoader(0, new Bundle(), this);
   }
}
```

Fragment

Fragment

- 複数の目的
  - 使いまわすことのできる UI のまとまり
  - Activity のライフサイクル下で動く UI をもたないモジュール
- Activity のライフサイクルと同期して動く

Fragment


Fragment is added
onAttach()
onCreate()
onCreateView()
onActivityCreated()
onStart()
onResume()
Fragment is active
User navigates backward or fragment is removed/replaced
The fragment is added to the back stack, then removed/replaced
onPause()
onStop()
onDestroyView()
The fragment returns to the layout from the back stack
onDestroy()
onDetach()
Fragment is destroyed

UI をまとめる Fragment

- 大きな画面ではマルチペイン(タブレット)
- 小さな画面ではドロワーメニュー(ハンドセット)

UI を持たない Fragment

- setRetainInstance(true)
- Activity の再生成を超えてデータを保持したい場面で使う

Fragment

- Do
  - Fragment 生成時にデータを渡すときは setArguments を使う
- Don’t
  - コールバックを setter で設定する

Support Libraries

- 新しい API Level の API を古い API Level へバックポートするライブラリ
  - v4 Support Library, v7 AppCompat Library, etc...
- Fragment は Support Libraries に含まれるものを使う
  - バグフィクスが配信される

Starting from Lollipop…

Android 5.x (API Level 21 ~ 22)

- Material Design
- Android RunTime(ART)
- JobSchedulers and GcmNetworkManager

JobSchedulers and GcmNetworkManager

- アプリケーションのプロセス内でジョブを実行するためのフレームワーク
  - バックグラウンドで動作するさまざまな処理の実行をコントロールする
  - 状況を見てフレームワークがよしなに実行タイミングを見計らってくれる
    - 接続しているネットワークの種類(従量課金かどうか、など)
    - 前回失敗時からのバックオフ

JobSchedulers and GcmNetworkManager

- バッテリー消費を抑えるためのしくみ
  - バックグラウンドで動作しつづけることを避ける
  - ネットワーク接続が良くない状況で不必要に通信を試みることを避ける

Doze

- バッテリー消費を抑えるためのしくみ第二弾
  - 画面が付いていない・充電中でない時に発動するモード
  - 定期的にバックグラウンド処理を実行可能な時間が割り当てられる
    - メンテナンスウィンドウ
  - AlarmManager や JobScheduler, SyncAdapter の処理も影響を受ける
  - WakeLock も無視

Doze Enhancements

- Marshmallow の Doze はデバイスを持ち歩く間は発動しなかった
- デバイスが静止中に Doze が発動すると CPU もスリープするようになった
  - WakeLock
  - AlarmManager
  - GPS/WiFi Scan

Background Limits

- Execution Limits と Location Limits
  - Execution: Service と Broadcast の制限
    - IntentService も影響を受ける -> JobIntentService
  - Location: 位置情報の制限

旧来の API の抽象化

- RecyclerView
- Architecture Components

RecyclerView

- ListView や GridView の抽象化
  - より柔軟に Adapter が生成する View をレイアウトできる
  - ViewHolder パターン
  - SnapHelper

ArchitectureComponents

- ライフサイクルの抽象化
  - Activity や Fragment, Service など -> LifecyclerOwner
  - ライフサイクルに連動すべきクラス -> LifecycleObserver
  - UI に紐づくデータを管理するクラス -> ViewModel

ArchitectureComponents

- ライフサイクルの抽象化
  - 変化し続けるデータとライフサイクルの紐付け -> LiveData
  - 大きなデータセットのページネーション -> Paging Library

ArchitectureComponents

```
public class MainActivity extends AppCompatActivity {
  private MainViewModel mViewModel;
  private TextView mTextView;

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mTextView = findViewById(R.id.hello_world);

    mViewModel = ViewModelProviders.of(this).get(MainViewModel.class);

    mTextView.setText(mViewModel.helloWorld());
  }
}
```

ArchitectureComponents

```
public class MainActivity extends AppCompatActivity {
  private MainViewModel mViewModel;
  private TextView mTextView;

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mTextView = findViewById(R.id.hello_world);

    mViewModel = ViewModelProviders.of(this).get(MainViewModel.class);

    mTextView.setText(mViewModel.helloWorld());
  }
}
```

ArchitectureComponents

```
public class MainActivity extends AppCompatActivity {
  private MainViewModel mViewModel;
  private TextView mTextView;

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mTextView = findViewById(R.id.hello_world);

    mViewModel = ViewModelProviders.of(this).get(MainViewModel.class);

    mTextView.setText(mViewModel.helloWorld());
  }
}
```

ArchitectureComponents

```
public class MainViewModel extends AndroidViewModel {
  public MainViewModel(Application application) {
    super(application);
  }

  public String getSomething() {
    return “Hello World!”;
  }

  @Override
  public void onCleared() {
    super.onCleared();
    Log.d(“MainViewModel”, “onCleared!”);
  }
}
```

ArchitectureComponents

```
public class MainViewModel extends AndroidViewModel {
  public MainViewModel(Application application) {
    super(application);
  }

  public String getSomething() {
    return “Hello World!”;
  }

  @Override
  public void onCleared() {
    super.onCleared();
    Log.d(“MainViewModel”, “onCleared!”);
  }
}
```

ArchitectureComponents

```
public class MainViewModel extends AndroidViewModel {
  public MainViewModel(Application application) {
    super(application);
  }

  public String getSomething() {
    return “Hello World!”;
  }

  @Override
  // Automatically called when Activity is destroyed
  public void onCleared() {
    super.onCleared();
    Log.d(“MainViewModel”, “onCleared!”);
  }
}
```

ArchitectureComponents

```
public class MainViewModel extends AndroidViewModel {
  public MainViewModel(Application application) {
    super(application);
  }

  public String getSomething() {
    return “Hello World!”;
  }

  @Override
  // Automatically called when Activity is destroyed
  public void onCleared() {
    super.onCleared();
    Log.d(“MainViewModel”, “onCleared!”);
  }
}
```

まとめ

まとめ

- API のベースは 1.0 の頃からほぼおなじ
  - 画面や非同期処理のライフサイクルはほぼ変わっていない
    - Activity, Fragment
    - Thread with Handler -> AsyncTask -> AsyncTaskLoader
  - 端末の進化にともなってアプリも進化しているので省メモリは今なお重要
    - JobScheduler and GcmNetworkManager

まとめ

- ライフサイクルの複雑さと付き合う方法が増えてきた
  - 非同期処理の抽象化: AsyncTask, AsyncTaskLoader
  - ライフサイクルの抽象化: ArchitectureComponents
  - View の抽象化: RecyclerView
- RxJava やそれをベースにした Flux など、代替手段もある

Keishin Yokomaku (@KeithYokoma) - DevFest Tokyo 2017

Retro/Prospective Android Application Development