Building apps using CustomViews

Keishin Yokomaku @ Drivemode, Inc.
potatotips #28

@KeithYokoma

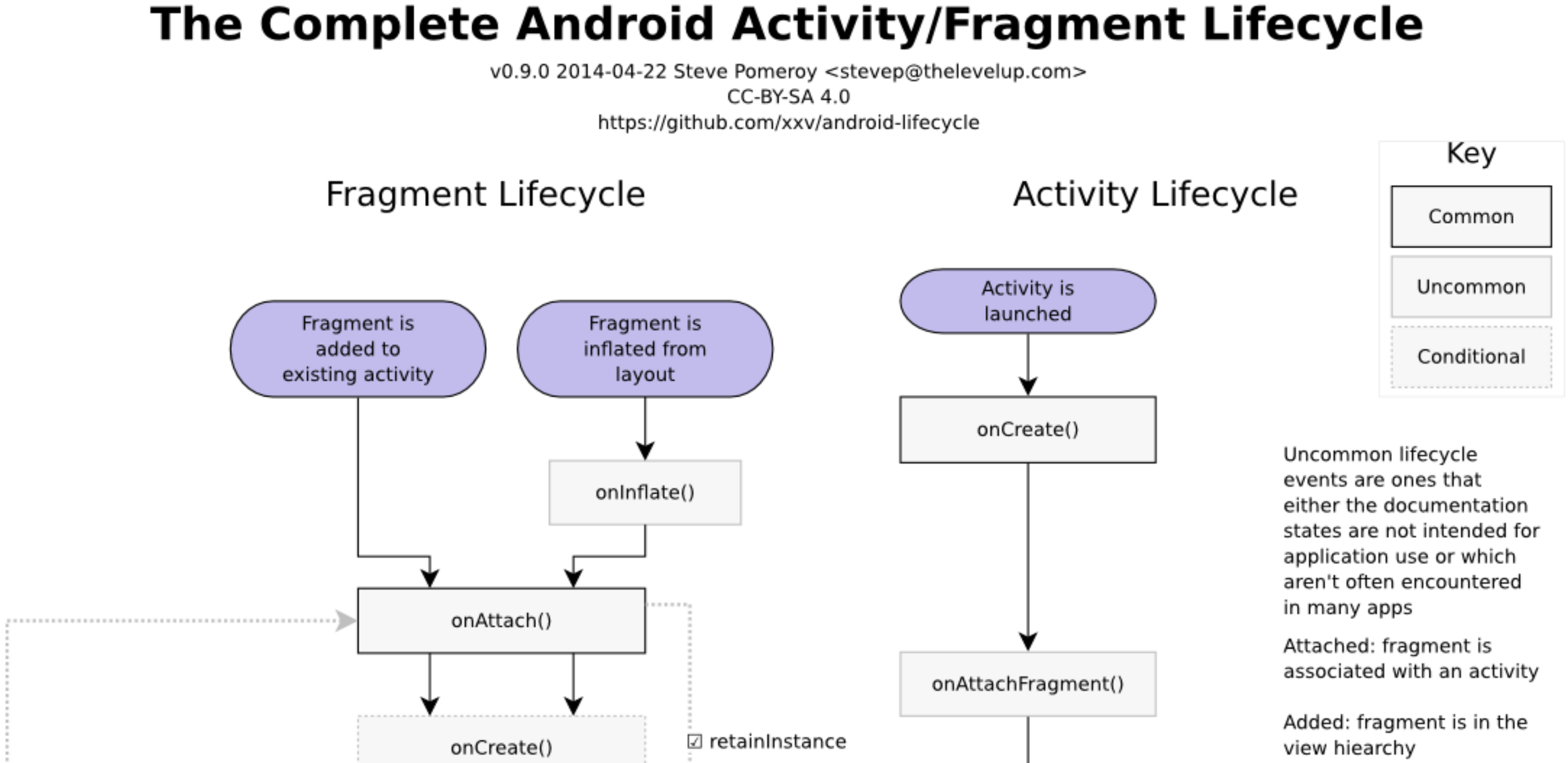
- Keishin Yokomaku at Drivemode, Inc.
- Work
  - Android apps
  - Android Training and its publication
- Like
  - Bicycle, Photography, Tumblr and Motorsport
- AIDL は友達

Why not Fragments?

Why not Fragments?

- Too much complexity
- Hard to debug, hard to test

Too much complexity



Hard to debug, hard to test

- Asynchronous operations using FragmentManager/FragmentTransaction.
- Tightly coupled with Views.
- View and business logic can be placed in a fragment.
- But view logic can be placed in CustomViews.
  - If we decouple business logic from Fragment, what’s left in Fragment?

“I’m deep in fragment spaghetti, how do I escape?”

–@Piwai at Square, Inc.

Escape from spaghetti💨

- Advocating Against Android Fragments from Square, Inc.
  - It’s pointless to have Fragments which are hard to test and debug.
  - “Use Presenter to isolate business logic into dedicated controllers.”
  - “Presenter makes the code more readable and facilitates testing.”

“kk. I got CustomViews is the way to go, but how?”

–You

Ways to go with CustomViews

- View-based frameworks
  - Mortar and Flow
  - Scoop
  - Rosie
  - Conductor
  - screenplay
  - and so on…

The Square way

- Flow
  - manages back stack of the screen flow
  - executes transition between screens
- Mortar
  - isolates dagger modules for each screens

The Square way

Activity

PathContainer

- Transition between screens(Path)
- Back stack management

Path

- Declares a screen
- Holds a Presenter to work with CustomView
- Lifecycle management of CustomView(save states to bundle and restore them from bundle, etc…)

CustomView

- Contains view logic

Path and Presenter

```
@Layout(R.layout.screen_sample)
@WithModule(SampleScreen.Module.class)
public class SampleScreen extends Path {
  private final Something something;

  public SampleScreen(Something something) {
    this.something = something;
  }

  @dagger.Module(injects = {SampleView.class}, complete = false)
  public class Module {
    @Provides Something provideSomething() {
      return something;
    }
  }

  @Singleton
  public static class Presenter extends ViewPresenter<SampleView> {
    private final Something something;

    @Inject Presenter(Something something) {
      this.something = something;
    }
  }
}
```

Path and Presenter

```
@Layout(R.layout.screen_sample)
@WithModule(SampleScreen.Module.class)
public class SampleScreen extends Path {
  private final Something something;

  public SampleScreen(Something something) {
    this.something = something;
  }

  @dagger.Module(injects = {SampleView.class}, complete = false)
  public class Module {
    @Provides Something provideSomething() {
      return something;
    }
  }

  @Singleton
  public static class Presenter extends ViewPresenter<SampleView> {
    private final Something something;

    @Inject Presenter(Something something) {
      this.something = something;
    }
  }
}
```

Path and Presenter

```
@Layout(R.layout.screen_sample)
@WithModule(SampleScreen.Module.class)
public class SampleScreen extends Path {
  private final Something something;

  public SampleScreen(Something something) {
    this.something = something;
  }

  @dagger.Module(injects = {SampleView.class}, complete = false)
  public class Module {
    @Provides Something provideSomething() {
      return something;
    }
  }

  @Singleton
  public static class Presenter extends ViewPresenter<SampleView> {
    private final Something something;

    @Inject Presenter(Something something) {
      this.something = something;
    }
  }
}
```

Path and Presenter

```
@Layout(R.layout.screen_sample)
@WithModule(SampleScreen.Module.class)
public class SampleScreen extends Path {
  private final Something something;

  public SampleScreen(Something something) {
    this.something = something;
  }

  @dagger.Module(injects = {SampleView.class}, complete = false)
  public class Module {
    @Provides Something provideSomething() {
      return something;
    }
  }

  @Singleton
  public static class Presenter extends ViewPresenter<SampleView> {
    private final Something something;

    @Inject Presenter(Something something) {
      this.something = something;
    }
  }
}
```

Path and Presenter

```
@Layout(R.layout.screen_sample)
@WithModule(SampleScreen.Module.class)
public class SampleScreen extends Path {
  private final Something something;

  public SampleScreen(Something something) {
    this.something = something;
  }

  @dagger.Module(injects = {SampleView.class}, complete = false)
  public class Module {
    @Provides Something provideSomething() {
      return something;
    }
  }

  @Singleton
  public static class Presenter extends ViewPresenter<SampleView> {
    private final Something something;

    @Inject Presenter(Something something) {
      this.something = something;
    }
  }
}
```

Path and Presenter

```
public class SampleScreen extends Path {
  /* emitted */

  @Singleton
  public static class Presenter extends ViewPresenter<SampleView> {
    private final Something something;

    @Inject Presenter(Something something) {
      this.something = something;
    }

    @Override public void onLoad(Bundle savedInstanceState) {
      super.onLoad(savedInstanceState);
      if (!hasView()) return;
      getView().setSomething(something);
    }
  }
}
```

Path and Presenter

```
public class SampleScreen extends Path {
  /* emitted */

  @Singleton
  public static class Presenter extends ViewPresenter<SampleView> {
    private final Something something;

    @Inject Presenter(Something something) {
      this.something = something;
    }

    @Override public void onLoad(Bundle savedInstanceState) {
      super.onLoad(savedInstanceState);
      if (!hasView()) return;
      getView().setSomething(something);
    }
  }
}
```

Path and Presenter

```
public class SampleScreen extends Path {
  /* emitted */

  @Singleton
  public static class Presenter extends ViewPresenter<SampleView> {
    private final Something something;

    @Inject Presenter(Something something) {
      this.something = something;
    }

    @Override public void onLoad(Bundle savedInstanceState) {
      super.onLoad(savedInstanceState);
      if (!hasView()) return;
      getView().setSomething(something);
    }
  }
}
```

Path and Presenter

```
public class SampleScreen extends Path {
  /* emitted */

  @Singleton
  public static class Presenter extends ViewPresenter<SampleView> {
    private final Something something;

    @Inject Presenter(Something something) {
      this.something = something;
    }

    @Override public void onLoad(Bundle savedInstanceState) {
      super.onLoad(savedInstanceState);
      if (!hasView()) return;
      getView().setSomething(something);
    }
  }
}
```

CustomView

```
public class SampleView extends FrameLayout {
  @Inject SampleScreen.Presenter presenter;

  public SampleView(Context context, AttributeSet attrs) {
    super(context, attrs);
    ObjectGraphService.inject(context, this);
  }

  @Override
  protected void onAttachedToWindow() {
    super.onAttachedToWindow();
    presenter.takeView(this);
  }

  @Override
  protected void onDetachedFromWindow() {
    presenter.dropView(this);
    super.onDetachedFromWindow();
  }

  public void setSomething(Something something) { /* something */ }
}
```

CustomView

```
public class SampleView extends FrameLayout {
  @Inject SampleScreen.Presenter presenter;

  public SampleView(Context context, AttributeSet attrs) {
    super(context, attrs);
    ObjectGraphService.inject(context, this);
  }

  @Override
  protected void onAttachedToWindow() {
    super.onAttachedToWindow();
    presenter.takeView(this);
  }

  @Override
  protected void onDetachedFromWindow() {
    presenter.dropView(this);
    super.onDetachedFromWindow();
  }

  public void setSomething(Something something) { /* something */ }
}
```

CustomView

```
public class SampleView extends FrameLayout {
  @Inject SampleScreen.Presenter presenter;

  public SampleView(Context context, AttributeSet attrs) {
    super(context, attrs);
    ObjectGraphService.inject(context, this);
  }

  @Override
  protected void onAttachedToWindow() {
    super.onAttachedToWindow();
    presenter.takeView(this);
  }

  @Override
  protected void onDetachedFromWindow() {
    presenter.dropView(this);
    super.onDetachedFromWindow();
  }

  public void setSomething(Something something) { /* something */ }
}
```

CustomView

```
public class SampleView extends FrameLayout {
  @Inject SampleScreen.Presenter presenter;

  public SampleView(Context context, AttributeSet attrs) {
    super(context, attrs);
    ObjectGraphService.inject(context, this);
  }

  @Override
  protected void onAttachedToWindow() {
    super.onAttachedToWindow();
    presenter.takeView(this);
  }

  @Override
  protected void onDetachedFromWindow() {
    presenter.dropView(this);
    super.onDetachedFromWindow();
  }

  public void setSomething(Something something) { /* something */ }
}
```

The Square way

```
/* How to move to SampleScreen */

// create some object that you would like to pass to the next screen
Something something = new Something();

// instantiate SampleScreen with arguments
Flow.get(getContext()).set(new SampleScreen(something));
```

The Square way

```
/* How to move to SampleScreen */

// create some object that you would like to pass to the next screen
Something something = new Something();

// instantiate SampleScreen with arguments
Flow.get(getContext()).set(new SampleScreen(something));
```

👍 and 👎

✓ Activity can be Service

- Flow and Mortar work on Overlay views

⚠ Still, you need to be careful on the view lifecycle

- Not fully escaped from spaghetti of asynchronous things
- Use `hasView()` to check if View is detached or not

✗ Not enough MaterialDesign support

- No Shared Element transition support

Building apps using CustomViews

Keishin Yokomaku @ Drivemode, Inc.
potatotips #28