Click to View: What Happens?

Question:

> Explain the processes involved from clicking a launcher icon to displaying a layout on screen. For example, user clicks the application icon on the launcher and is displayed with a "Hello, World" text on the middle of a white screen - how does the text get there? Your answer should cover: Intents, Application, Activity, View lifecycles

The launcher

The firs process involved here is actually completely up to the launcher. Even the fact that there was an icon in the first place the courtesy of this (underappreciated) piece of software. Here is an example of what the open-source Essential launcher does when a user clicks an icon:

```
@Override
public void onClick(final View view) {
    if (view instanceof ImageView
        && view.getTag() instanceof ApplicationModel) {

        final ApplicationModel aM = (ApplicationModel) view.getTag();

        if (aM.packageName == null || aM.className == null) {
            return;
        }

        final Intent intent = IntentUtil
            .newAppMainIntent(aM.packageName, aM.className);

        if (IntentUtil.isCallable(context.getPackageManager(), intent)) {
            context.startActivity(intent);
        }
    }
}
```

In this case the `ApplicationModel` class is a POJO that contains information on the app, namely the label used for display (e.g. `MyApplication`), the package name (e.g. `com.example.app`) and the full class name of the activity being launched (e.g. `com.example.app.MainActivity`).

The static method `IntentUtil.newAppMainIntent` creates an intent using these data. It is customary for launchers to use the `Intent.ACTION_MAIN` as the action label. Typical applications register one activity to handle this action in the manifest. It also sets the `Intent.FLAG_ACTIVITY_NEW_TASK` flag. This flag

indicates to the Android system, that the launcher does not want to "own" any activities started with this intent, but that they should have their own history stack.

The is callable method here actually uses the `PackageManager` to check whether or not the application has a main intent, and whether the launcher has the proper permissions to start it. Applications have access to fine grained authorization controls on who can and can not start certain parts of them. This provides a certain level of security, and allows developers to reason about their applications to a point.

The `context.startActivity(intent)` call finalizes the operation by sending the intent to the system.

Initializing the Application

The Android system then proceeds to initialize the so called "package". A Java class loader will look up any Application subclass and the system will call it's no argument constructor. While nothing actually requires you to subclass the Application, non-trivial applications will have some sort of setup that needs to happen early in the lifecycle, for which the `onCreate` method is used (and sometimes abused). But this method will get called a little bit later.

The thing to happen next, will be the class loader looking up the classes for any Content providers and initializing them in the so-called priority order by calling their no-argument constructors. This may seem surprising, as traditionally these are not considered to be part of an application lifecycle. However, they are! As a matter of fact, the Firebase SDK uses them to make integration easy for anyone (as outlined in a Firebase blog post from 2016). Note, that technically, these already have access to the application object created previously.

It's now time for the `onCreate` method of the `Application` (sub)class to be called. It will do any work deemed absolutely critical by the developer. Of course, in a simple application that shows only a `Hello World!` string there won't be anything to do here.

Starting an Activity

Since intents are supposed to be handled by some component of an application, and because the `Intent.ACTION_MAIN` is typically handled by an activity, the next step in the process is to start an Activity. Selection of the activity will be performed as specified by the application manifest file. The system will call the constructor of an activity, which is usual

The activity `onCreate` method is called first in this case, the purpose of which is to assure that critical components on which the activity relies are all ready to go.

Developers are naturally urged to keep this snappy. The method may be called with a saved instance state later (for example after the device changes orientation, this is used to persist state). This is, naturally, an important consideration.

Typically, an implementation of this method calls the `setContentView` method which specifies a layout (or a view directly, as it doesn't necessarily have to be a layout) to be used as the visible part of the Activity. This spawns many views to be created as the layout is inflated (i.e. any views required by the theme of the activity such as window decorations and any views directly specified in the layout).

For each of these, the `onCreateView` method on the activity side is called. Note, that at this point the `onCreate` method hasn't returned yet. It's simply the internal calls of the `setContentView` method that require these views to be created as part of the layout inflation process. This process then calls the constructor of our text view, that will eventually, end up showing the `Hello World!` string of text. Finally, the layout inflation process is finished, and the `onCreate` method returns.

The activity has two more methods that developers can hook into: `onStart` and `onResume`. What should be placed in which has been a matter of debate, and design. What is clear, is that the `onResume` method should be used only to continue any animations. The `onStart` method can be used to perform any dynamic view configuration if necessary.

The activity is then ready to attach to a window provided by the system. This will be done by the UI thread looper, which will eventually (and hopefully soon) decides to draw a frame (in a stacktrace this will be signified by the looper calling `doFrame`). This is signaled to an activity by calling the `onAttachedToWindow` method when it is ready do draw the first frame. This will propagate to all of the views in hierarchy, as inflated previously.

The TextView Does Work

At this point one may want to rush to go through the view lifecycle. However, then we would miss the text. And the key question was where does the text come from! For this we need to step back to the layout inflation process, and this time look at it from the perspective of a `View` object.

Anyone, who has ever tried to implement a custom view knows the pain of dealing with 4 overloaded constructors. The simplest one only has a `Context` argument, while the full one additionally has a set of attributes specified to the inflater (think simple XML attributes), and two parameters used for styling. If our text string is static, it will most likely be specified in an XML layout file and you will find the value (a text string) in the attribute set. If the developers care about internationalization, it will most probably be placed in a string resource file and you will find a reference to this resource entry (an integer value) in this attribute

set, which would then be used to lookup the string from the application resources (which have been optimized for fast access). The constructor is responsible for guarding against any inconsistencies here and preparing the object for eventual drawing. Finally, when the inflation is complete, the `onFinishInflate` method of the view will be called.

When the frame is being drawn, all of the parent-views and children-views will have to negotiate for the oh-so-limited space on screen. This, as in most UI toolkits, is implemented by the parent asking the children to express their wishes on size by calling the `onMeasure` method. This method has arguments (as a fair parent would provide some arguments) which indicate the size suggested by the parent, but this can be rather lenient (perhaps the parent can resize itself if their children are bigger or smaller). During this call, a child must call the `setMeasuredDimension` with their decided measurements. This is a contract that can not be breached. Additional calls to `onMeasure` will be performed, until a consensus is reached in the view hierarchy (hopefully).

The `layout` method is then called, with the parents' decision, on where to place the child view element. Afterwards, the `onLayout` method is called, allowing a view to layout it's children if it has any. Finally, a call to the `draw` method is made. It is meant for manual rendering of the entire view, and in most subclasses of a `View` remain unused. Then the `onDraw` method will be called. This is probably *the* most important method of them all as far as view logic is concerned. It takes a `Canvas` object as an argument, which is an abstraction of the drawing mechanism of Android. The view will proceed to invoke methods of the `Canvas` as necessary, to make it appear as specified. In reality, the internals of a TextView class are rather complicated: there is a lot of math involved in typesetting.

When all of the views draw themselves onto a canvas, the frame is displayed.

```
Hello World!
```

What Got Left Behind?

- Anything static. Static variables are oftentimes used (for better or worse) by developers of Android applications. It is important to remember, that anything static will be evaluated as the class loader first loads the class. This is particularly important to consider as a point of discouragement, when considering to evaluate some computed value in a static variable. You may find yourself surprised when it actually happens (i.e. when the first import statement of that class is evaluated).
- Fragments! While this trivial example assumes that no fragments are used, in reality this is one of the most important building blocks of the Android system and introduce another level of complexity.
- Processes and threads. For the purpose of this presentation, I deemed this to be out-of-scope, but they are something to consider. After all,

applications are Linux processes (or collections of processes) at the end of the day.