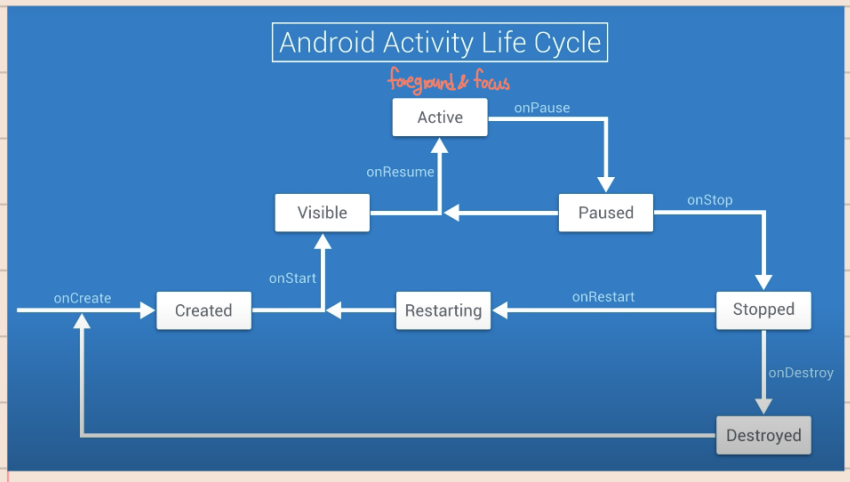
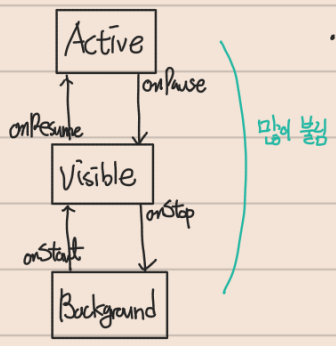
Android Activity Lifecycle





- onCreate() / onDestroy()는 app running 동안 1번만 불림

Quiz

What is the order that the lifecycle events are called **after** the device is rotated.

✓ onPause, onStop, onDestroy, onCreate, onStart, onResume

Nice work! Rotating the device causes the Activity to be destroyed and recreated, so our lifecycle starts at onPause and ends at onResume. Note that we don't see onRestart, which only happens if the activity is stopped (but not destroyed) and then restarted.

- Device orientation / screen width 런타임에 변경될 수 있음
- default behavior: destroy & recreate whenever a device configuration changes.

Save & Restore Instance State

- onSaveInstanceState이 언제불리는지 : onSaveInstanceState() → onStop() → onDestroy()

Our solution is imperfect -- We're missing onStop and onDestroy, because those states happen after onSaveInstanceState is called . We at least see the lifecycle all the way through onSaveInstanceState and you'll see that the TextView gets filled with more logs as you rotate the device more.

```
private static final String LIFECYCLE_CALLBACKS_TEXT_KEY = "callbacks";
```

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    logAndAppend(ON_SAVE_INSTANCE_STATE);
    String lifecycleDisplayTextViewContents = mLifecycleDisplay.getText().toString();
    outState.putString(LIFECYCLE_CALLBACKS_TEXT_KEY, lifecycleDisplayTextViewContents);
}
```

· onCreate() 에서 기존의 저장된 상태 불러오기

```
if (savedInstanceState != null) {
    if (savedInstanceState.containsKey(LIFECYCLE_CALLBACKS_TEXT_KEY)) {
        String allPreviousLifecycleCallbacks = savedInstanceState
                .getString(LIFECYCLE_CALLBACKS_TEXT_KEY);
        mLifecycleDisplay.setText(allPreviousLifecycleCallbacks);
    }
}
```

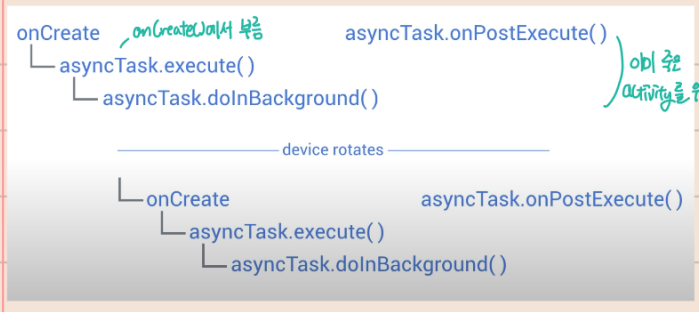
Preparing for Termination

· Android does everything it can to make the resource limitations of the device Invisible to user.

· From a system perspective, onPause and onStop are signals that our app may be killed imminently

AsyncTask and Loaders

· AsyncTask lifecycle



Problem

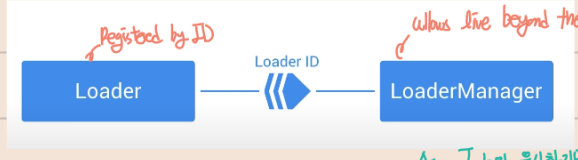
· Background 스레드가 결과를 activity의 한 부분으로 보내야 함 → AsyncTask가 죽은 activity를 keep (스레드가 종료될 때까지)

extra memory pressure

Solution: Loader

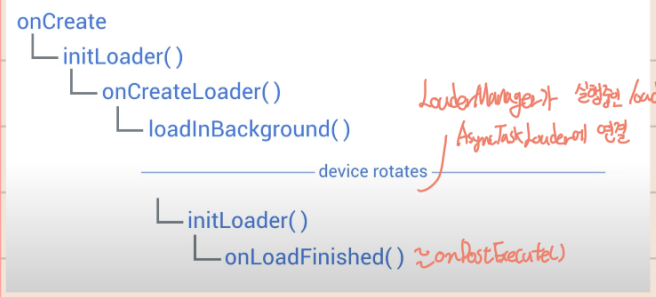
· Provide a framework to perform asynchronous loading of data

Preventing duplicate loads from happening in parallel



AsyncTask와 유사하지만 lifecycle이 다름

· Background thread에서 data를 load하고 싶으면 : AsyncTaskLoader라는 loader Pattern 사용



장점

① Retrieving the result to the current active activity

② Preventing duplication of background threads

③ Helping to eliminate duplication of zombie activities.

Code

```
// (1) implement LoaderManager.LoaderCallbacks<String> on MainActivity
@RequiresApi(api = Build.VERSION_CODES.HONEYCOMB)
public class MainActivity extends AppCompatActivity implements LoaderManager.LoaderCallbacks<String> {
```

```
// (2) Create a constant int to uniquely identify your loader. Call it GITHUB_SEARCH_LOADER
private static final int GITHUB_SEARCH_LOADER = 17;
```

· Interface method override

```
@SuppressLint("StaticFieldLeak")
@Override
public Loader<String> onCreateLoader(int i, final Bundle bundle) {
    return new AsyncTaskLoader<String>( context: this) {
        @Override
        public String loadInBackground() {
            // (10) Get the String for our URL from the bundle passed to onCreateLoader
            String searchQueryUrl = bundle.getString(SEARCH_QUERY_URL_EXTRA);
            if (TextUtils.isEmpty(searchQueryUrl)) {
                return null;
            }
            String githubSearchResults = null;
            try {
                URL githubSearchUrl = NetworkUtils.buildUrl(searchQueryUrl);
                githubSearchResults = NetworkUtils.getResponseFromHttpUrl(githubSearchUrl);
            } catch (IOException e) {
                e.printStackTrace();
            }
            return githubSearchResults;
        }

        @Override
        protected void onStartLoading() {
            if (bundle == null) {
                return;
            }
            mLoadingIndicator.setVisibility(View.VISIBLE);
            forceLoad();
        }
    };
}
```

· onCreate()에서 initLoader(). 맨처음에 null로 init.

```
// (24) Initialize the loader with GITHUB_SEARCH_LOADER as the ID, null for the bundle, and this for the callback
getSupportLoaderManager().initLoader(GITHUB_SEARCH_LOADER,  args: null,  callback: this);
```

· makeGithubSearchQuery() 함수 안에서 또 다시 init()

```
// (19) Create a bundle called queryBundle
Bundle queryBundle = new Bundle();
// (20) Use putString with SEARCH_QUERY_URL_EXTRA as the key and the String value of the URL as the value
queryBundle.putString(SEARCH_QUERY_URL_EXTRA, githubSearchUrl.toString());
// (21) Call getSupportLoaderManager and store it in a LoaderManager variable
android.support.v4.app.LoaderManager loaderManager = getSupportLoaderManager();
// (22) Get our Loader by calling getLoader and passing the ID we specified
Loader<String> loader = loaderManager.getLoader(GITHUB_SEARCH_LOADER);
// (23) If the Loader was null, initialize it. Else, restart it.
if (loader == null) {
    loaderManager.initLoader(GITHUB_SEARCH_LOADER, queryBundle,  callback: this);
} else {
    loaderManager.restartLoader(GITHUB_SEARCH_LOADER, queryBundle,  callback: this);
}
```

· They are designed to reload if the user navigates away from the activity and then returns.
↳ caching과 redelivering으로 필요없는 reload 방지가능

Caching

· onCreateLoader()의 return시 anonymous class 안에서!

```
// (1) Create a String member variable called mGithubJson that will store the raw JSON
private String mGithubJson;
```

· onStartLoading() 안에서

```
// (2) If mGithubJson is not null, deliver that result. Otherwise, force a load
if (!TextUtils.isEmpty(mGithubJson)) {
    deliverResult(mGithubJson);
} else {
    mLoadingIndicator.setVisibility(View.VISIBLE);

    forceLoad();
}
```

- anonymous class 안에서 override.

```
@Override
public void deliverResult(String data) {
    mGithubJson = data;
    super.deliverResult(data);
}
```