Week 8: Android Lesson

[Refer to the Lesson2 Lecture Note

Agenda

- Java: Static Factory Method
- Java: Builder Design Pattern
- Logcat
- Explicit Intent
- Implicit Intent
- Android Activity Life Cycle
- Saving app data using SharedPreferences
- Unit Test in Java

- Cohort Class: Build Currency Converter App

OO Design Principles

Recall some principles you have learnt before:

- Single Responsibility
- Open/Closed Principles
- Encapsulate what varies
- Favour composition over inheritance
- Program to a supertype

Java: Static Factory Method

- A static factory method is a method in a class definition that returns an instance of that class.
- Constructor can be declared private to limit/simplify how the object is instantiated

Java: Static Factory Method

Example

```
public class Tea {
    private boolean sugar;
    private boolean milk;

    Tea( boolean sugar, boolean milk){
        this .sugar = sugar;
        this .milk = milk;
    }
    public static Tea teh(){
        return new Tea( true , true );
    }
    public static Tea tehkosong(){
        return new Tea( false , true );
    }
}
```

Invoke the static method to instantiate a predefined object

```
public static void main(String[] args) {
    Tea tea1 = Tea.tehkosong();
}
```

Java: Builder Design Pattern

```java
public class TeaTwo {
    private boolean sugar;
    private boolean milk;
    private boolean ice;
    private boolean toGo;
}
```

Problem:
You need to write a LOT of constructors to include all possible combinations

Solution:
Builder Design Pattern

Steps:
1. Set the constructor to be **private** (so user cannot instantiate using **new** keyword)
2. Create a **static nested class** as the builder
3. Inside the builder, create **method** to set each **attribute**
4. A method to **return** the actual object

Java: Builder Design Pattern

Builder Design Pattern

```java
public class TeaTwo {
    private boolean sugar;
    private boolean milk;
    private TeaTwo(TeaBuilder teaBuilder){
        this.sugar = teaBuilder.sugar;
        this.milk = teaBuilder.milk;
    }

    static class TeaBuilder {
        private boolean sugar;
        private boolean milk;

        TeaBuilder(){}

        public TeaBuilder setSugar ( boolean sugar){
            this .sugar = sugar;
            return this; }

        public TeaBuilder setMilk ( boolean milk){
            this .milk = milk;
            return this; }

        public TeaTwo build (){
            return new TeaTwo( this ); }
    }
}
```

Exercise:
Add two more attributes (e.g. ice & toGo), then modify the builder class as well

How the builder is used

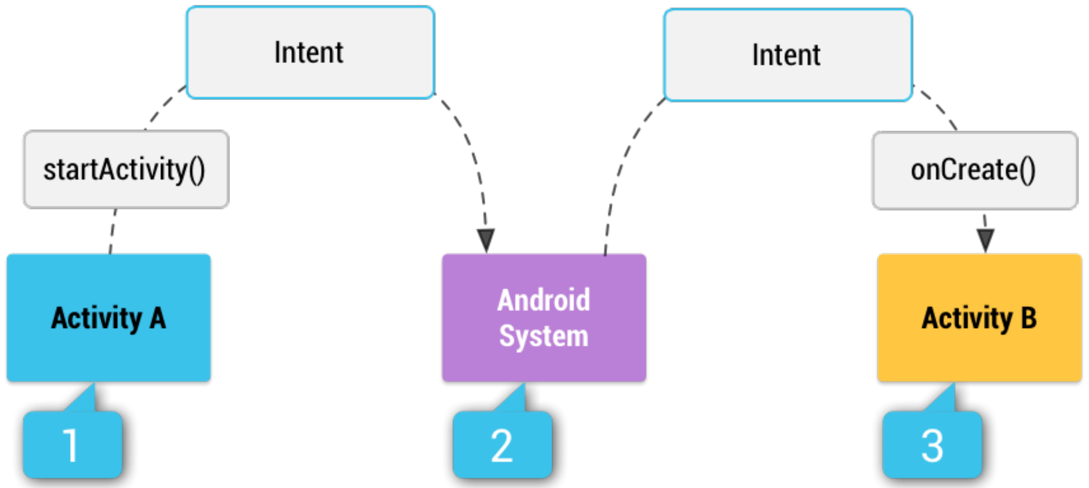
```java
TeaTwo teaTwo = new TeaTwo.TeaBuilder().setSugar(true).setMilk(true).build()
```

Logcat

- Using Log instead of printing into console helps programmer not only to **debug,** but also to **search, filter, or save** logging messages.
- Use ***Log*** class in Android utility package to display log message
- There are 4 logging levels:
  - ***d*** for debug
  - ***i*** for info
  - ***w*** for warning
  - ***e*** for error

Intent

- is a messaging object you can use to request an action from other activities or apps



https://developer.android.com/guide/components/intents-filters

Explicit Intent

- to start another **specific** activity
- The code below creates an Intent object and then starts the *SubActivity* activity from the current activity (*MainActivity)*

```
Intent intent = new Intent(MainActivity.this, SubActivity.class);
startActivity(intent);
```

- Data can be passed by using *putExtra()* method

```
Intent intent = new Intent(MainActivity.this, SubActivity.class);
intent.putExtra(KEY,value);
startActivity(intent);
```

- In *SubActivity*, you can obtain the data this way

```
Intent intent = getIntent();
double value = intent.getDoubleExtra(MainActivity.KEY,
defaultValue);
```

Implicit Intent

- Instead of starting a specific activity, implicit intent is used by declaring a **general action** to perform, which allows a component from **another app** to handle it.
- For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.
- Some common intents are Calendar, Map, Camera, File Storage, Email, etc

https://developer.android.com/guide/components/intents-common

https://developer.android.com/guide/components/intents-filters#Types

Implicit Intent

- Example

```
String location = "SUTD";
Uri.Builder builder = new Uri.Builder();
builder.scheme( "geo" ).opaquePart( "0.0" ).appendQueryParameter( "q" ,location);
Uri geoLocation = builder.build();

Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(geoLocation);
if (intent.resolveActivity(getPackageManager()) != null) {
    startActivity(intent);
}
```

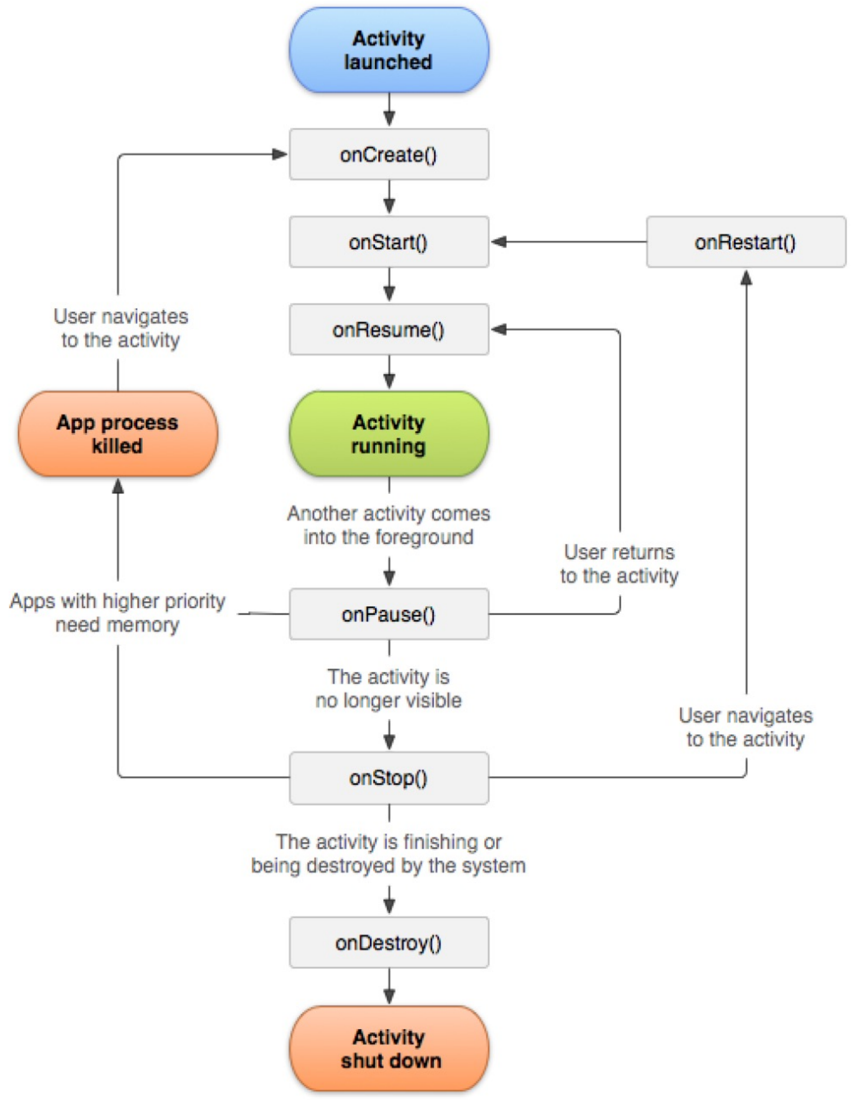
Add the following to the activity component in manifest file

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="geo" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Invoking the Intent above will open a map app

URI = Uniform Resource Identifier. You can think of it as the ”address” of resource

Android Activity Life Cycle


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Saving app data using SharedPreferences

- Shared preferences allow you to store small amounts of primitive data as key/value pairs in a file on the device. To get a handle to a preference file, and to read, write, and manage preference data, use the SharedPreferences class. The Android framework manages the shared preferences file itself. The file is accessible to all the components of your app, but it is not accessible to other apps.

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-4-saving-user-data/lesson-9-preferences-and-settings/9-1-c-shared-preferences/9-1-c-shared-preferences.html

Saving app data using SharedPreferences

- Example

```
private final String sharedPrefFile = "com.example.android.myapplication" ;
public static final String KEY = "MyKey" ;
SharedPreferences mPreferences;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    //other code not shown
    mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
    String Rate_text = mPreferences.getString(KEY, "Default Value");
}

@Override
protected void onPause () {
    super.onPause();
    SharedPreferences.Editor preferencesEditor = mPreferences.edit();
    preferencesEditor.putString(KEY, "Value to be saved");
    preferencesEditor.apply();
}
```

Unit Test in Java

Why do unit testing?

- Unit testing validates that your software works, even in the face of continual changes in your code
- Writing code for unit testing also forces your program to be modular
- JUnit4 is a commonly-used open-source framework to conduct unit testing.
- To conduct unit testing, you write your tests in a test class.

Unit Test in Java

Why do unit testing?

- Unit testing validates that your software works, even in the face of continual changes in your code
- Writing code for unit testing also forces your program to be modular
- JUnit4 is a commonly-used open-source framework to conduct unit testing.
- To conduct unit testing, you write your tests in a test class.