Lesson 3 - Comic App

Objectives

- Describe how concurrency is implemented using Executor, Runnable, Handler and Looper
- Query an API using the Executor and Runnable
- Describe F-bounded Polymorphism
- Apply Template Method Design Pattern
- Modify the android Manifest to set permissions and fix the orientation
- Download JSON data given a URL of an API call
- Parse JSON data using the JSONObject class
- Download an Image file given a URL
- Display the information in the UI

Introduction

In this lesson, we are going to download data from the xkcd.com web API and display the comic image in our app.

The Android/Java that you need to know

Abstract Classes

A class declared as **`abstract`** cannot be instantiated.

Typically, **abstract classes** have **abstract methods**, which you will recall is a **method signature** that includes the keyword abstract.

Why do we do this?

- **What is to be done** is specified by the abstract methods (and interfaces too)
- **How it is to be done** is specified by their implementation

Because abstract classes can behave as a datatype, it promotes **code reuse**.
Recall the principle of **Program to a supertype**: You may write methods that take in an abstract class as an input, and you are guaranteed that objects passed to it will have the abstract method implemented. (This remark also applies to java interfaces.)

Your abstract class can have constructors specified. One use case is if you would like to force the initialization of any instance variables.
Remember that constructors are not inherited.
Thus, in the constructor of the child classes, you will need to call the superclass constructor using the **`super()`** keyword.

To use an abstract class, sub-class it and implement any abstract methods.

Clicker Question

Given the following

```
abstract class Foo {
   int x = 0;                                    // (i)
   abstract void f() { System.out.println(x); } // (ii)
   void g();                                     // (iii)
   abstract void h();                            // (iv)
}
```

Which statement(s) cause(s) compilation errors?

In the example below, complete the class **`Tiger`**.

```
public class TestAbstract {

    public static void main(String[] args){
        Feline tora = new Tiger("Tiger","Sumatran Tiger");
        makeSound(tora);
    }

    public static void makeSound(Feline feline){
        feline.sound();
    }
}

abstract class Feline {

    private String name;
    private String breed;

    public Feline(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }

    public String getName() {
        return name;
    }

    public String getBreed() {
        return breed;
    }

    public abstract void sound();
}

class Tiger extends Feline{

     //write the constructor and complete the class
}
```

Template Method Design Pattern

One application of an abstract class is in the **template method design pattern**.
In this design pattern, there is an algorithm with a fixed structure, but the implementation of some steps are left to the subclasses. The following example is taken from "Heads First Design Patterns - A brain-friendly guide".

We have a fixed way of brewing caffeine beverages (Coffee, Tea etc), but you'll agree that how you brew it and what condiments to add depends on the beverage.
In the example below,

- The algorithm to make the caffeine beverage, **`prepareRecipe()`** is declared **`final`** to prevent subclasses from altering the algorithm.
- The steps in the algorithm that are common to all beverages are implemented.
- The steps that can vary are declared **`abstract`**.

```
public abstract class CaffeineBeverage {

    final void prepareRecipe(){
        boilWater();
        brew();
        addCondiments();
        pourInCup();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater(){
        System.out.println("Boiling Water");
    }

    void pourInCup(){
        System.out.println("Pouring in Cup");
    }
}
```

We may then have our subclasses of **`CaffeineBeverage`**.

```
class GourmetCoffee extends CaffeineBeverage{
    @Override void brew() {
        System.out.println("Put in Coffee Maker");
    }

    @Override void addCondiments() {
        System.out.println("Adding nothing, because GourmetCoffee");
    }
}
```

We may then brew our coffee:

```
CaffeineBeverage caffeineBeverage = new GourmetCoffee();
caffeineBeverage.prepareRecipe();
```

Generic Classes & Interfaces

You would have used the ArrayList class in the following way::

```
ArrayList<Integer> arrayList = new ArrayList<>();
```

Recall that this design is for **type safety** - an operation cannot be performed on an object unless it is valid for that object.
This means that the code below would cause a compile-time error.

```
arrayList.add("abc");
```

You would also have worked with the Comparable interface:

```
public class Octagon implements Comparable<Octagon>{
    //code not shown
    @Override
    public int compareTo(Octagon octagon) {
        //code not shown
    }
}
```

The **`ArrayList`** class is an example of a **Generic class** where a class takes in an object or objects as a parameter and operates on it.
Similarly, the **`Comparable<T>`** interface is an example of a **Generic interface**.

When Generic meets Inheritance

Recall the **`Pair`** class that we introduced during the Java Revision lesson.

```
public class Pair<T, S> {
  public T first;
  public S second;
  public Pair(T first, S second)
  { this.first = first; this.second = second; }
}
```

And the **`Comparable`** interface, which you have learnt.

```
interface Comparable<T> {
   int compareTo(T that);
}
```

Suppose we would like to modify the above **`Pair`** class such that when two **`Pair`** objects are compared, we compare the **`first`** items, then if there is a tie, we go on to compare the **`second`** items.

However, that would imply that the generic **`T`** and **`S`** are *not as generic* as before, because in order for **`Pair`** to be **`Comparable`**, you would need **`T`** and **`S`** to be **`Comparable`** too (otherwise you would not know how to compare **`T`** and **`S`**).

Thus, Java allows us to place constraints on the Type parameters too. These are called **Bounded Type Parameters**.

We modify the **`Pair`** class as follows.

```
public class Pair<T extends Comparable<T>, S extends Comparable<S>>
   implements Comparable<Pair<T,S>> {
  public T first;
  public S second;
  public Pair(T first, S second)
  { this.first = first; this.second = second; }
  @Override
  public int compareTo(Pair<T,S> that) {
      int r1 = this.first.compareTo(that.first);
      if (r1 == 0) {
          return this.second.compareTo(that.second);
      } else {
          return r1;
      }
  }
}
```

In the above the type constraint **`T extends Comparable<T>`** restricts the type variable `T` must be a subtype of **`Comparable<T>`**. These constraints provide hints to the compiler that it is safe to call **`this.first.compareTo(that.first)`**. This is known as F-bounded polymorphism.

Further Reading:

- https://docs.oracle.com/javase/tutorial/java/generics/bounded.html
- https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html

A glimpse of concurrent programming

In many situations, we want our applications to handle different tasks simultaneously. In this app, we are going to query a web API to retrieve data.

The length of time this task will take depends

- on the internet connection,
- file size, etc

The Android Framework has one main thread for the UI.

While the data is being retrieved, you want the UI to remain responsive and inform the user that the download is still in progress. You'll realise that one thing the programmer cannot control is the speed of the user's internet connection.

This means that the **download should be done on a separate thread from the UI.**

A thread is a unit of executing instruction sequences in a program.

A concurrent program consists of multiple threads and an **executor (scheduler)** which orchestrates the actual execution/scheduling of the threads. In the presence of multiple processing units (CPU cores), the scheduler might be able to execute/schedule multiple threads simultaneously in different processing units, otherwise, the scheduler will interleave the executions of the given set of scheduled threads. This is often abstracted away from the programmers, so as to prevent deadlock or other abnormal outcomes.

Question: Do you know what is the difference between a thread and a process?

The Executor class

In Java, we gain access to the executor service / the scheduler service via the Executor class. Through the executor, we are able to access the pool of threads for a program.

The Runnable interface

An instance of the Runnable interface denotes a sequence of instructions to be executed in a thread.

```
// main thread (i.e. UI thread)
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
        @Override
        public void run() {
            // a new thread
            // some instructions to be executed in the new thread.
        }
});
```

`Executors.newSingleThreadExecutor()` - this instantiates a single thread executor service.
Other construction methods are available, e.g. **`newFixedThreadPool(int nThreads)`**. See the documentation for details.
`run()` - this is an abstract method defined in the **Runnable** interface. An instance of **Runnable** interface must implement/override this method.

Further Reading

- https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html
- Y. Daniel Liang, “Introduction to Java Programming”, 10th Edition. Chapter 30.

Example of how Executor and Runnables are Used

In the code below (Adapted from Liang, Chapter 30):

- Calling the **`newSingleThreadExecutor()`** static method allocates one thread only
- By calling **`executorService.execute()`** four times, we get four processes
- Since there is only one thread, the four processes are run in sequence.

If you change the ExecutorService object to

```
ExecutorService executorService = Executors.newFixedThreadPool(4);
```

this specifies four threads and the output suggests that the processes are run in parallel.

```
public class MyClass {

   String ss;

   public static void main(String[] args) {

       int n = 20;
       ExecutorService executorService = Executors.newSingleThreadExecutor();
       String s = "abcd";
       for(int i = 0; i < s.length(); i++ ){
           executorService.execute(
                new PrintStr( String.valueOf( s.charAt(i) ) , n));
       }
       executorService.shutdown();
   }
}

class PrintStr implements Runnable{

   String s;
   int times;

   PrintStr(String s, int times){
       this.s = s;
       this.times = times;
   }

   @Override
   public void run() {

       for(int i = 0; i < times; i++){
           System.out.print(s + i + " ");
       }
       System.out.println();
   }
}
```

Immutable variable

Question: What happens if the Runnable / the sub-routine running in the thread needs to exchange information with the main thread (UI thread)?
Instructions in the child thread can only access variables from the main thread if they are immutable (i.e. final).

The following is resulting in a compilation error.

```
// main thread (i.e. UI thread)
int s = 0;
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
                    // a new thread
                    @Override
                    public void run() {
                        s = s + 1; // illegal access
                        System.out.println(s);
                    }
                });
```

This is ok.

```
// main thread (i.e. UI thread)
final int s = 0;
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
                    // a new thread
                    @Override
                    public void run() {
                        System.out.println(s);
                    }
                }
```

But it implies that **shared data cannot be modified** in the child thread.

Making use of a generic class

We can introduce a generic class **Container** which serves as a workaround.

```
// a container class
class Container<T> {
    T value;
    Container(T v) { this.value = v; }
    void set(T v) { this.value = v; }
    T get() { return this.value; }
}
```

We put the data to be exchanged between the main and child threads in a **Container** object.

```
// main thread (i.e. UI thread)
String s = ...;
final Container<String> cs = new Container<>(s);
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
        @Override
        public void run() {
            // a new thread
            String s1 = cs.get() + "!";
            cs.set(s1);
        }
});
```

Question: How can we inform the UI thread when the child thread is done *asynchronously*? In other words, without having the UI thread to constantly check whether the child thread is done.

Running tasks on Android's main thread

In the Android multithreading library, every thread is associated with a message queue. A message queue is created for the purpose of asynchronous communication. (Just like our messagers, email, WhatsApp, Telegram, for interpersonal communication, without disturbing each other).

There are two additional classes in the Android Framework created for the purposes of reading and updating the message queues: the **Looper** class and the **Handler** class.

Let us illustrate this with how things work on the main thread. Android has a main thread (also called the UI thread) that is used to process events e.g. button clicks etc. These events or Messages are stored in a **`MessageQueue`**.

With this main thread, you have a **`Looper`** object, which works with the **`MessageQueue`** to send **`Message`** objects to the main thread. These messages can be **`Runnable`** objects too. In other words, the **`Looper`** class manages the **`MessageQueue`** and this is abstracted away from you.

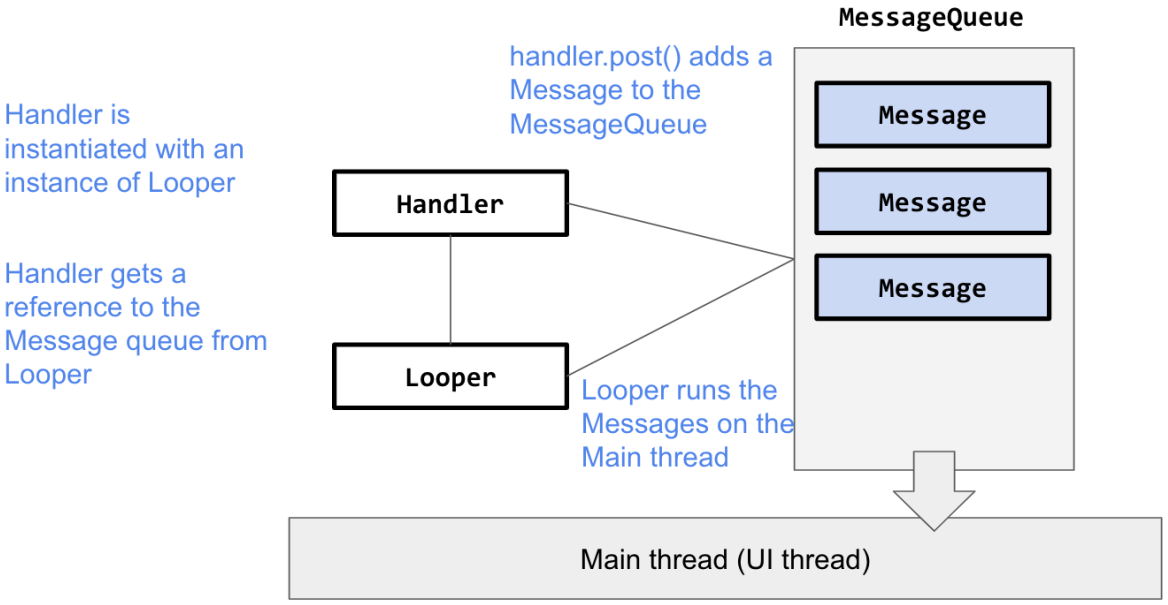
You use a **`Handler`** object when you need to have greater control and send your own messages to be run on the Main thread.
First, instantiate the **`Handler`** object and pass an instance of the main Thread's **`Looper`**.

```
Looper uiLooper = Looper.getMainLooper(); // get the main looper
final Handler handler = new Handler(uiLooper); // get the handler
```

Next, when you are ready to send a task to the main thread, call **`handler.post()`**. Write your custom task within the anonymous **`Runnable`** object.

```
handler.post(new Runnable(){...} );
```

The following diagram summarises the relationships.



Further reading

- Documentation for the Handler, Looper, Message Queue classes
  - https://developer.android.com/reference/android/os/Handler.html
  - https://developer.android.com/reference/android/os/MessageQueue
  - https://developer.android.com/reference/android/os/Looper
- You could also dig into the source code and see exactly how it works, and verify the accuracy of my description above
  - https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/Handler.java
  - https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/Looper.java
- Many blog posts discuss this issue too.
  - https://medium.com/@iamatul_k/deep-dive-in-handler-message-messagequeue-and-looper-7ee10eab4716

How to use Looper and Handler with your background thread

Thus, to have your background thread talk to the main UI thread, this is what happens.

1. First, you instantiate a **`Handler`** object and pass an instance of the main thread's **`Looper`** to it. At this stage, Handler obtains a reference to the **`MessageQueue`**.
2. Execute whatever instructions you like in your background thread.
3. When your background thread is ready to execute some instructions on the main thread, call **`handler.post()`** and pass your instructions in a new **`Runnable`** object to it. **`handler.post()`** adds your **`Runnable`** object to the UI thread's **`MessageQueue`**.
4. The main thread's **`Looper`** then manages the message queue and decides when to execute your **`Runnable`** object on the UI thread. This is abstracted away from you.

```
// main thread (i.e. UI thread)
ExecutorService executor = Executors.newSingleThreadExecutor();
Looper uiLooper = Looper.getMainLooper(); // get the main looper
final Handler handler = new Handler(uiLooper); // get the handler for
the main thread
executor.execute(new Runnable() {
        @Override
        public void run() {
            // a new thread
            // instructions performed in the child thread
            // ...
            handler.post(new Runnable() {
                @Override
                public void run() {
                    //UI Thread will receive and run this
                }
            });
        }
});
```

Notice now that you need two **`Runnable`** instances

- The “outer Runnable” is for the background thread
- The ”inner Runnable” is for the background thread to talk to the UI thread

How to use Executor and Runnable in your Android App

Step 1. Prevent your activity from changing orientation.

- In the Android Activity Lifecycle, recall that the app layout is destroyed and re-created if the screen is rotated.
- If the child thread is running in the background during this process, it will not be able to display the result on the re-created activity. (c.f. Phone number changes)
- Hence, you need to constrain your activity so that it is always in your desired orientation.
- This is done in the android manifest.

Step 2. Define a generic class in MainActivity or some util package which serves as a container.

Make the following decisions to help you decide the generic types:

- What information launches the background task (the child thread) ?
- What information do I want to give the user as the task proceeds?
- What information does the background task provide?
- When the background task is completed, how shall the UI be updated?

Step 3. Decide what jobs are to be run in the two Runnable instances:

- **`run()`** `in the outer` **`Runnable`** always carries out the background task (child thread) e.g. downloading the image data from a URL.
- **`run()`** `in the inner` **`Runnable`** carries out the job to be performed (in the UI thread) after the background task is complete. Suppose the task is to download an image given a URL, this is where you write code to display the image on the UI.

Step 4. Define a method getComic() which implements the background task as well as the handler task (task to be performed upon completion of the background task).

One possible code stump is shown below. In this code stump, **Look at the method signatures and think about why the final type is required. Observe how the generic type containers are used.**

```
void getComic(final String userInput) {
    ExecutorService executor = Executors.newSingleThreadExecutor();
    final Handler handler = new Handler(Looper.getMainLooper());

    executor.execute(new Runnable() {
        @Override
        public void run() {
            //Background work here
            final Container<Bitmap> cBitmap = new Container<>();
            //retrieve the bitmap from xkcd and store it in cBitmap
            handler.post(new Runnable() {
                @Override
                public void run() {
                    //UI Thread work here
                    if (cBitmap.get() != null) {
                        // retrieve the bitmap and display it
                    }
                }
            });
        }
    });
}
```

Step 5. Decide how the user is to execute the background task.

Let's say your user will download the image with a button click. Then you would put the following code in the anonymous class within **`setOnClickListener()`**

```
getComic(userInput);
```

An Alternative - Fun Challenge

Can we extract the reusable part from the `getComic()` code and write a modular code by using generic?

See below for solution:

```
abstract class BackGroundTask<I,O> {
   ExecutorService executor;
   final Handler handler = new Handler(Looper.getMainLooper());
   public BackGroundTask() {
       this.executor = Executors.newSingleThreadExecutor();
   }
   abstract public O runInBackground(I i);
   abstract public void whenDone(O o);

   public void run(final I i) {
       final Container<O> co = new Container<>();
       this.executor.execute(new Runnable() {
           @Override
           public void run() {
               co.set(BackGroundTask.this.runInBackground(i));
               handler.post(new Runnable() {
                   @Override
                   public void run() {
                       if (co.get() != null) {
                           whenDone(co.get());
                       }
                   }
               });
           }
       });
   }
}
```

We may now define a nested class to replace `getComic()` by extending the `BackgroundTask` class.

```
void getComic2 (final String userInput) {
    (new BackGroundTask<String, Bitmap>() {
        @Override
        public Bitmap runInBackground(String userInput) {
            Bitmap bitmap = null;
            // background task here
            return bitmap;
        }

        @Override
        public void whenDone(Bitmap bitmap) {
            if (bitmap != null) {
                // UI thread job here
            }
        }
    }).run(userInput);
}
```

Using the xkcd.com web API

Many websites provide an API for you to access the data. In this lesson, we are going to use the xkcd.com API to access the comics.

This is all the documentation that is provided, in the **About** section:

> **Is there an interface for automated systems to access comics and metadata?**
>
> Yes. You can get comics through the JSON interface, at URLs like http://xkcd.com/info.0.json (current comic) and http://xkcd.com/614/info.0.json (comic #614).

Hence, this is the easiest web API that I could find and we are going to use it.

Building a URL

A URL is a URI that refers to a particular website. The parts of a URL are as follows

Component	Example
Scheme	`https`
Authority	`xkcd.com`
Path	`info.0.json` or `614/info.0.json`

To prevent parsing errors you may use the **`Uri`** builder before creating a **`URL`** object.

Using such a builder helps to eliminate coding errors.

The string constants may be placed in the **`strings.xml`** file instead.

After you create the **`Uri`** object, use it to make a **`URL`** object.

The constructor of the **`URL`** class throws a **`MalformedURLException`**.

As this is a **checked exception**, you have to put the code in a **`try-catch`** block.

```
final String scheme = "https";
final String authority = "xkcd.com";
final String back = "info.0.json";
URL url = null;

Uri.Builder builder = new Uri.Builder();
builder.scheme(scheme)
       .authority(authority)
       .appendPath(back);

Uri uri = builder.build();

try{
   url = new URL(uri.toString());
}catch(MalformedURLException ex) {
   Log.i(TAG, "malformed URL: " + url.toString());
}
```

Connecting to the internet

With a URL object, you are ready to download data from the internet.
In the starter code, you are provided with a **`Utils`** class.
You need not worry about how they are implemented, although if you are querying an API for your 1D project, you might find the code useful.
You should see that some of these methods print data to the logcat.

The following method takes in a URL for an image and returns the image as a Bitmap object. Please note the exceptions thrown by this method.

```
static Bitmap getBitmap(URL url)
```

The following method takes in a Context object and checks if a network connection is available. True is returned if a network connection is available, False if is not. **`Context`** is the superclass of **`AppCompatActivity`**.

```
static boolean isNetworkAvailable(Context context)
```

> If you use the Android Emulator, you may experience difficulties with checking for an internet connection. Please borrow a phone.

Android Manifest

In order to

- Constrain the activity orientation
- Access the internet

The android manifest needs some additional information.

They are shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.norman_lee.comicapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="ComicApp"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity"
            android:screenOrientation="portrait"
            android:configChanges="orientation|keyboardHidden">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

</manifest>
```

The JSON format

JSON stands for **J**ava**S**cript **O**bject **N**otation and stores data using key-value pairs.

A sample response from the xkcd API is


```
{"month": "11", "num": 2068, "link": "", "year": "2018", "news": "",
"safe_title": "Election Night", "transcript": "", "alt": "\"Even the
blind\u00e2\u0080\u0094those who are anxious to hear, but are not able
to see\u00e2\u0080\u0094will be taken care of. Immense megaphones have
been constructed and will be in use at The Tribune office and in the
Coliseum. The one at the Coliseum will be operated by a gentleman who
draws $60 a week from Barnum & Bailey's circus for the use of his
voice.\"", "img": "https://imgs.xkcd.com/comics/election_night.png",
"title": "Election Night", "day": "5"}
```


Before you make use of this data, you would have to make sense of it. Online JSON viewers are available to help you.

Parsing JSON data using the JSONObject class

Once you are able to make sense of this data, you are ready to parse it. One way is to use the **`JSONObject`** class.

The **`JSONObject`** constructor takes in a string variable that contains the JSON data.

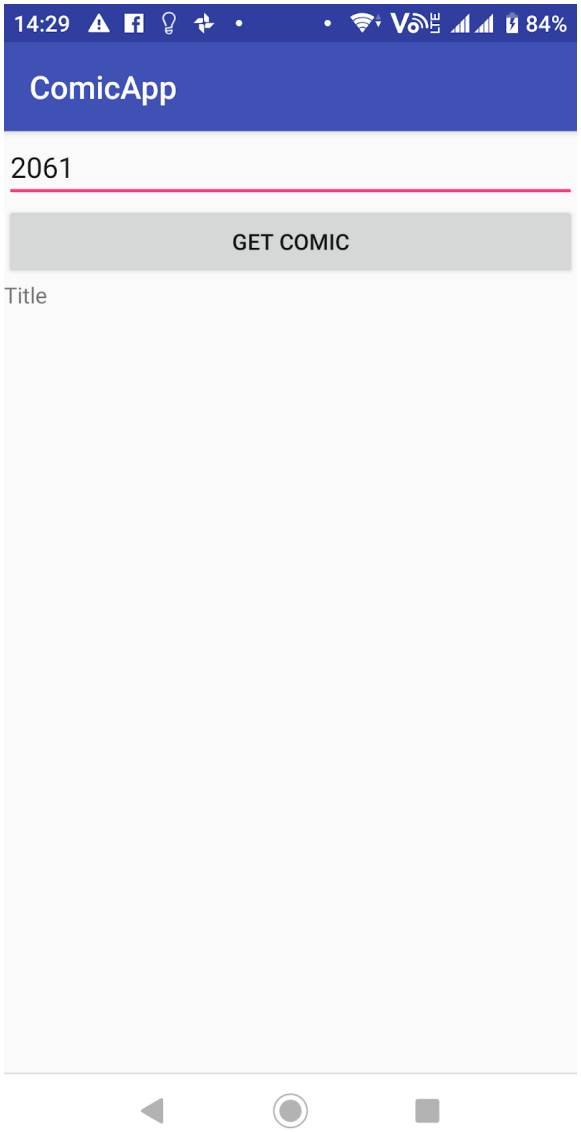
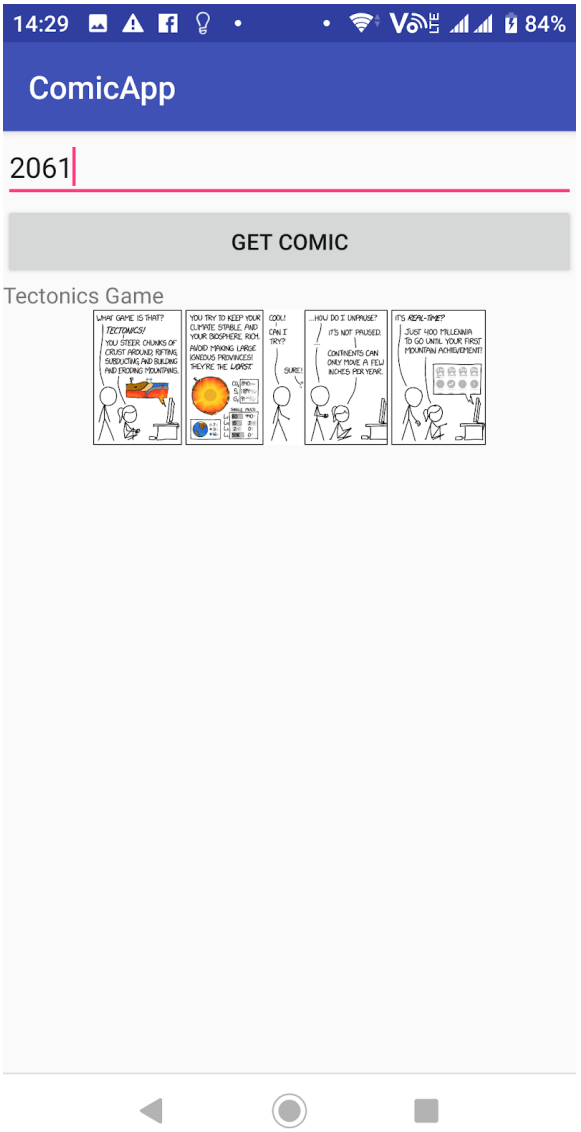
You then retrieve the value using the key and one of the appropriate get methods.

```
JSONObject jsonObject = new JSONObject(json);
String safe_title = jsonObject.getString("safe_title");
```

Another alternative is the GSON library, which many programmers find useful in parsing JSON data. I will leave you to learn it on your own and you won't be tested on it.

Completing Your App

What the app should do

Step 1. A user enters a number from 1 to the latest xkcd comic number and clicks **Get Comic**.	**Step 2.** The comic and its **URL** is displayed. (sorry image below is not updated)
	

BEFORE YOU BEGIN, download the starter code and compile your app. You may need to Build → Clean Project.

TODO 6.0 Study the Utils class and see what methods are available for you

TODO 6.1 Ensure that Android Manifest has permissions for internet and has the orientation fixed

TODO 6.2 - 6.5 When button is clicked, Get the user input

- Get references to widgets
- Set up setOnClickListener for the button
- Retrieve the user input from the EditText
- If network is active, call the getComic method. Otherwise, show a Toast message

TODO 6.6 - 6.15 Complete the getComic method implementation with Executor, Runnable, Looper and Handler to download the comic

- Make sure an executor and a handler are instantiated
- (inside worker thread) make use of the Container class from Utils. The container is useful for sharing object between the main thread and the child thread
- Call Utils.buildURL to get the URL based on the comic number from userInput
- Call Utils.getJSON to get the String response of the URL
- If the response is null, write a Toast message in main thread, otherwise:
- Inside a try/catch, get JSON object from the String response
- Extract the image string url with key "img" from the JSON object
- Extract the title with key "safe_title"
- Download the Bitmap using Utils.getBitmap
- (main thread) Assign the bitmap downloaded to imageView and set the title to textViewTitle.

TODO 6.16 (Challenge) Re-do steps 6.6 - 6.15 by making use of the `BackgroundTask` abstract class. (Hints: you may consider applying the template method design pattern)

Possible Solution:

```
abstract class BackgroundTask<I,O> {
    abstract public O task(I input);
    abstract public void done(O result);
    public void start(final I userInput) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        final Handler handler = new Handler(Looper.getMainLooper());

        executor.execute(new Runnable() {
            @Override
            public void run() {
                //Background work here
                final Container<O> c = new Container<O>();
                O o = task(userInput);
                c.set(o);

                handler.post(new Runnable() {
                    @Override
                    public void run() {
                        //UI Thread work here
                        if (c.get() != null) {
                            done(c.get());
                        }
                    }
                });
            }
        });
    }
}
```