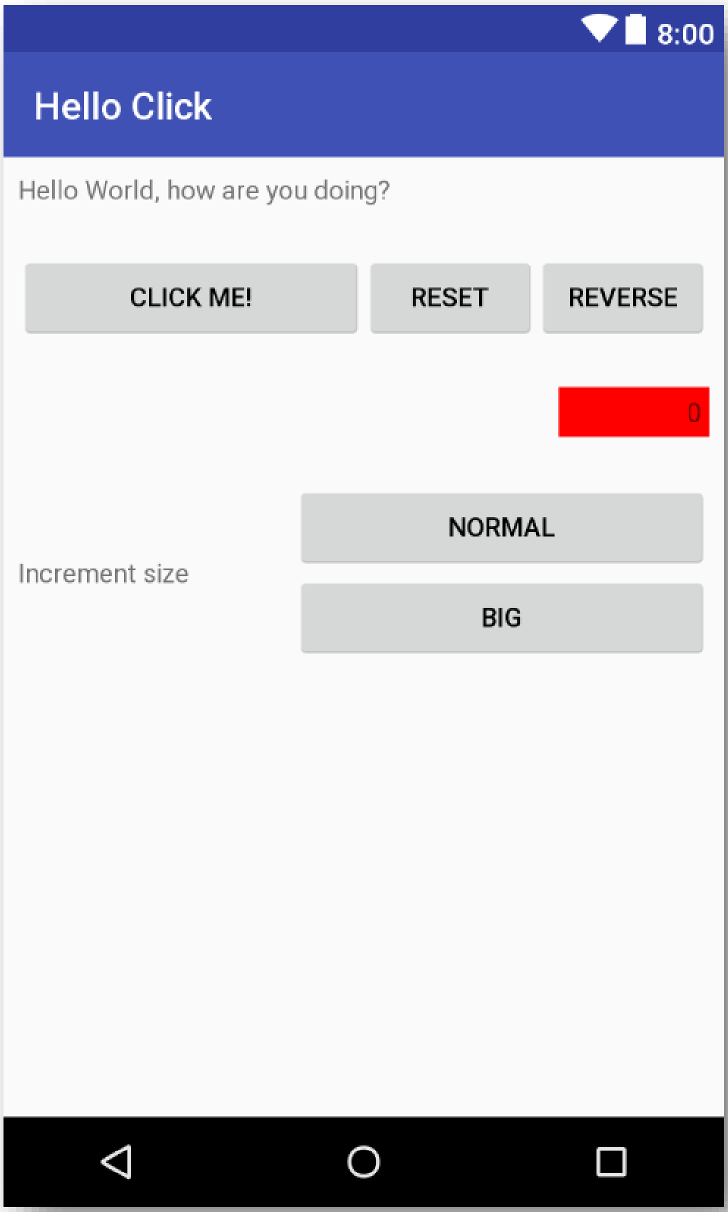
Android: Session Two

At this point, you should have the Hello Click application UI finished:



Part 1: Catch your click

In this part, we are going to finish the app. It has a complete UI but no code to actually "do" anything.

To catch the click on our buttons, we have to set an event listener on them, but first things first, you need to get a reference to the button that is inside your "activity_main.xml".

We can bind references on UI elements from a layout file, provided they have an ***id*** attribute by using Kotlin Android Extensions. As its name implies, this is a set of tools that extend Kotlin language capabilities more specifically for Android.

Then use the imported bound reference to attach a listener to the click event on the button by using the setOnClickListener().

Somewhere in *activity_main.xml* layout resource file…

```
<Button
    android:id="@+id/main_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="I am the button"/>
```

In *MainActivity.kt* source file imports...

```
// Imports references for all widgets with an id from activity_main.xml
import kotlinx.android.synthetic.main.activity_main.*
```

In *MainActivity.kt* ***onCreate()*** callback function…

```
main_button.setOnClickListener {
    Log.d("MainActivity", "Main button was clicked !")
}
```

OnClickListener activity

Since the setOnClickListener() method can accept either a lambda expression - like in the example above – or an object implementing the View.OnClickListener interface, we often use the following alternate version of implementing the interface directly on our Activity, and using our Activity instance as a listener, thus optimizing by having only one listener:

```kotlin
class MainActivity : AppCompatActivity(), View.OnClickListener {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Use Activity instance as a listener for both our buttons
        main_button.setOnClickListener(this@MainActivity)
        secondary_button.setOnClickListener(this@MainActivity)
    }

    override fun onClick(clickedView: View?) {
        if (clickedView != null) {
            when (clickedView.id) {
                R.id.main_button -> {
                    // Let us log something we can see in the logcat console
                    Log.d("MainActivity", "Main button was clicked !")
                }
                R.id.secondary_button -> {
                    // Let's show a Toast
                    Toast.makeText(
                            this@MainActivity,
                            "Sub button was clicked !",
                            LENGTH_SHORT).show()
                }
                else -> {
                    // In theory, you should never reach this line
                }
            }
        }
    }
```

Nullables

By default, variables cannot be null. If required, a variable can be marked as nullable by appending a “?” (question mark) after its type:

```
// This does not work
val toto : String = null
// This is ok
val toto : String? = null
```

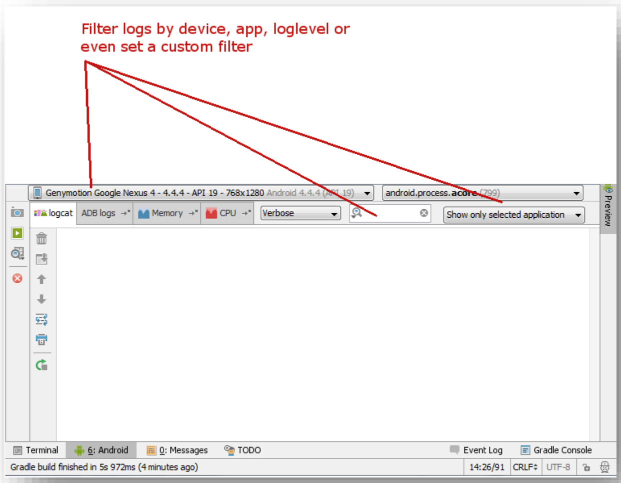
In a function, a nullable parameter can be enforces as not being null by appending “!!” (double exclamation marks) to its name. Be cautious though, only use when you are sure.

```
fun someFunction(param: String?) {
    // Does not work : param can be null
    Log.d("MainActivity", param.length.toString())
    // Works, we swear param is no null
    // This is pushy, use !! only when indeed param cannot be null
    Log.d("MainActivity", param!!.length.toString())
}

fun someFunctionAgain(param: String) {
    // Works, param cannot be null
    Log.d("MainActivity", param.length.toString())
}
```

Logging

To display a message on the console, use the “Log.d” method to display a debug message. The method takes 2 String parameters: the first one is a message tag you can filter afterward, and the second one is the message you want to log.

You can see all the messages from your application in the LogCat console



Genymotion Google Nexus 4 - 4.4.4 - API 19 - 768x1280 Android 4.4.4 (API 19)

android.process.acore (799)

Verbose

Show only selected application

Gradle build finished in 5s 972ms (4 minutes ago)

14:26/91 CRLF UTF-8

The Toast

A Toast is a kind of self-disappearing temporary popup. It is used to display some information to the user without interfering with the interface or the workflow.

```
// Create a Toast
val toast = Toast.makeText(
        this@MainActivity,
        "Some text to pop !",
        Toast.LENGTH_SHORT)
// Display the Toast
toast.show()
```

The Toast requires a context to display. Many android components require a context, and most of the time, we will use our Activity instance as a context.

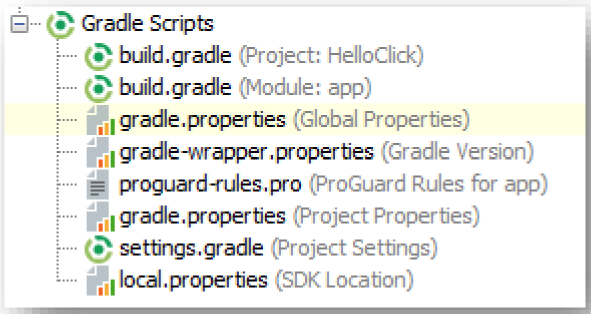
➪ Make the Hello Click application work 😉

- the “incrementButton” increments the counter
- the “clearButton” resets the counter
- the “normalButton” and “bigButton” switch the increment size
- the “reverseButton” make the counter decrease instead of increase

Part 2: Build

To build applications (compile code, compile byte-code, gather resources, generate apk, sign apk…) Android Studio uses Gradle build system.

When building the app, there are various Gradle scripts involved, which all take most of their building parameters in Gradle configuration files.
In a default project, there are 2 ***build.gradle*** files. One for the whole application, and one for the ***app*** module.

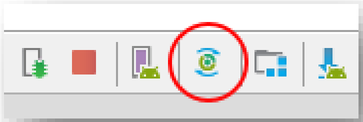

Gradle Scripts
build.gradle (Project: HelloClick)
build.gradle (Module: app)
gradle.properties (Global Properties)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

Gradle build files contain information that will be used at build time to create the final manifest file.

Now Gradle builds the app but is also used as a dependency manager. Libraries can be injected directly from distant repositories.

```
dependencies {
    // Includes automatically all jar files you put in a 'libs' directory
    implementation fileTree(dir: 'libs', include: ['*.jar'])

    // Imports kotlin, could prove useful eventually ;-)
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jre7:$kotlin_version"

    // AppCompat package, in charge of making recent Android features work
    // in older os versions
    implementation 'com.android.support:appcompat-v7:26.1.0'

    // The constraint layout we use is not part of the Android sdk
    implementation 'com.android.support.constraint:constraint-layout:1.0.2'

    // Testing stuff
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.1'
}
```

Modifications to this file must be notified to the Studio by clicking on this icon:



Part 3: invoking a webservice API

Now that we have a basic application running, let’s take it up to the next step and start using a WebService API to get some data.

To invoke methods from this WebService from Android, we could use the http stack included in the sdk, obtain some bytes, turning them into text, and then parse the text as structured data (xml, json…) to finally get useful objects. We could.

Or we can use a library to do it all for us. Plus, the default Android http stack is broken, so…

There are various libraries we can use to do this, like Volley, but one of the most flexible, powerful, focused and stable is Retrofit. So, this is the one we are going to use. Retrofit core lib is going to handle for us the interfacing with the WebService, and the asynchronous call. Under the hood, it relies on a great http stack called okhttp (which btw can also be used on its own as a – great - alternative to the default Android http stack). And to “translate” the response into real objects, format module plugins can be added to (almost) automatically translate html responses into objects.

Including Retrofit

To include Retrofit into your project, just at it as a dependency in your app ***build.gradle*** file.

```
// Import retrofit lib in our project
implementation 'com.squareup.retrofit2:retrofit:2.4.0'
// Also add a retrofit plugin to handle json responses
implementation 'com.squareup.retrofit2:converter-gson:2.4.0'
```

Allowing Internet access to the app

In Android, applications have rights and restrictions. In order to do some specific things, like connecting to the network, or accessing storage, or take pictures, an app must ask for permission.

Most permissions require user interaction to be obtained, but network access is considered a basic request in Android, so we can just ask for this permission by adding a declaration line in the manifest.

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

    <!-- Ask for the right to access the Internet -->
    <uses-permission android:name="android.permission.INTERNET"/>

    <application
        …
    </application>

</manifest>
```

Implementing Retrofit2

You can find the complete documentation to Retrofit at http://square.github.io/retrofit/ but here is a summary.

There are 4 main steps involved when working with Retrofit:

1. Create an interface to the WebService
2. Create a Retrofit component to instantiate a service client object implementing this Interface
3. Create a Callback object to handle the response
4. Pass Callback object to webservice function in service object to actually call the WebService

The following example works with the following sample WebService:

http://jsonplaceholder.typicode.com/todos?userId=2

Step 1: Create the Interface

```
interface WebServiceInterface {
    @GET("todos")
    fun listToDos(@Query("userId") userId: Int): Call<List<ToDoObject>>
}
```

Step 2: Create the Retrofit and service client objects

```
// A List to store or objects
val data = arrayListOf<ToDoObject>()
// The base URL where the WebService is located
val baseURL = "http://jsonplaceholder.typicode.com/"
// Use GSON library to create our JSON parser
val jsonConverter = GsonConverterFactory.create(GsonBuilder().create())
// Create a Retrofit client object targeting the provided URL
// and add a JSON converter (because we are expecting json responses)
val retrofit = Retrofit.Builder()
        .baseUrl(baseURL)
        .addConverterFactory(jsonConverter)
        .build()
// Use the client to create a service:
// an object implementing the interface to the WebService
val service: WSInterface = retrofit.create(WSInterface::class.java)
```

Step 3: The Callback object

```
val callback = object : Callback<List<ToDoObject>> {
    override fun onFailure(call: Call<List<ToDoObject>>?, t: Throwable?) {
        // Code here what happens if calling the WebService fails
        Log.d("TAG", "WebService call failed")
    }

    override fun onResponse(call: Call<List<ToDoObject>>?,
                            response: Response<List<ToDoObject>>?) {
        // Code here what happens when WebService responds
        if (response != null) {
            if (response.code() == 200) {
                // We got our data !
                val responseData = response.body()
                if (responseData != null) {
                    data.addAll(responseData)
                    Log.d("TAG", "WebService success : " + data.size)
                }
            }
        }

    }
}
```

Step 4: Call the WebService

```
// Finally, use the service to enqueue the callback
// This will asynchronously call the method
service.listToDos(2).enqueue(callback)
```

Part 4: Intents

Explicit intents

To go to another activity, first create an Intent in the starting activity :

```
// Create an explicit intent
val explicitIntent = Intent(activity, SecondActivity::class.java)

// Insert extra data in the intent
val message = "Hello from another world"
explicitIntent.putExtra("MESSAGE", message)

// Start the other activity by sending the intent
startActivity(explicitIntent)
```

And in the target activity, you can retrieve the content of the message:

```
// retrieve the intent that caused the activity to open
val originIntent = intent

// extract data from the intent
val message = originIntent.getStringExtra("MESSAGE")
```

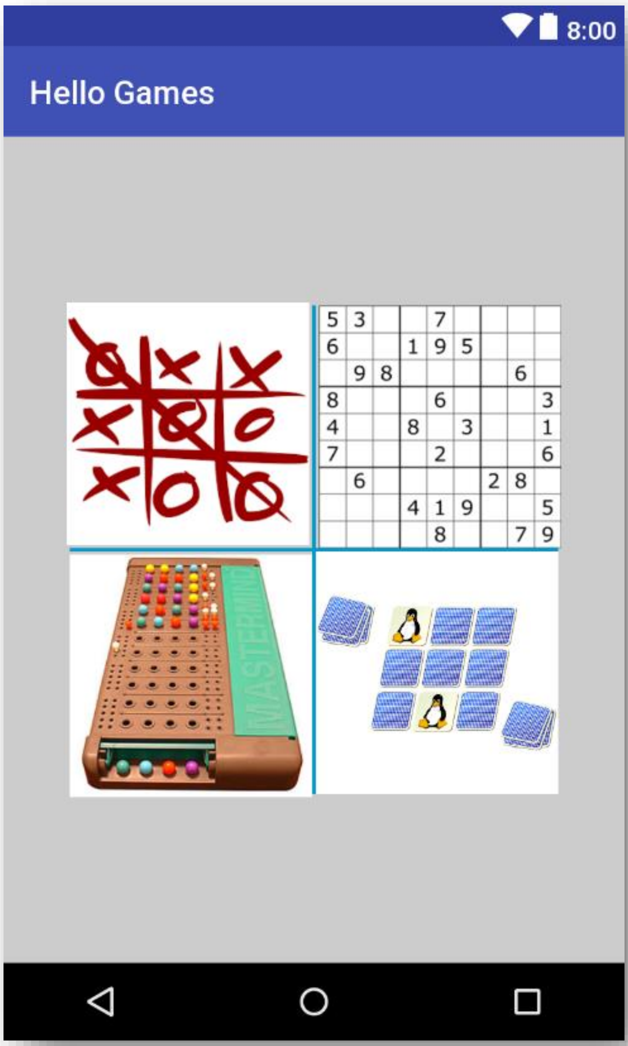
Implicit intents

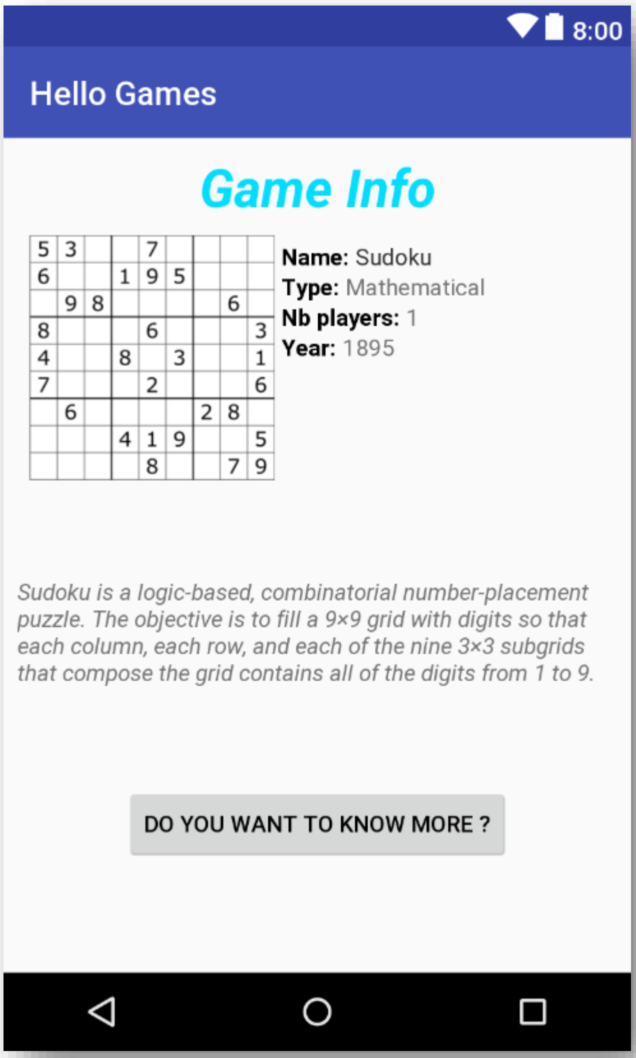
Some actions require an implicit Intent, since you do not know what activity to call:

```
val url = "http://www.epita.fr"
// Define an implicit intent
val implicitIntent = Intent(Intent.ACTION_VIEW)

// Add the required data in the intent (here the URL we want to open)
implicitIntent.data = Uri.parse(url)

// Launch the intent
startActivity(implicitIntent)
```

First submitted app

⇨ Create a new Hello Games application. This is a very simple application with only two screens:





Game Info

Name: Sudoku
Type: Mathematical
Nb players: 1
Year: 1895

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid contains all of the digits from 1 to 9.

The API we are going to work with can be found at the following address:

https://androidlessonsapi.herokuapp.com/api/help/

First screen, the game list. Get a list of games from the WebService. Pick 4 random games to display a simple menu as shown.
Upon clicking on a picture, open a second screen with details for the selected game.

Second screen, as you can see, game details. Upon clicking on the button, open a browser either on Google search for the game, or on the info url provided in game details.