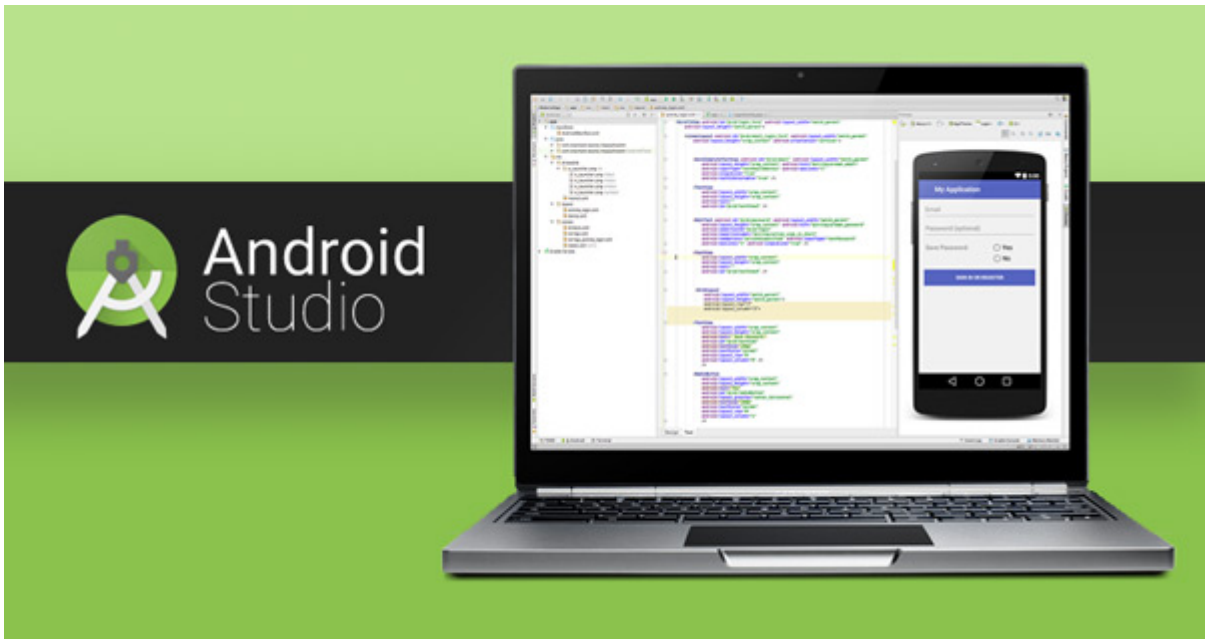
Configurando Android Studio



Introducción

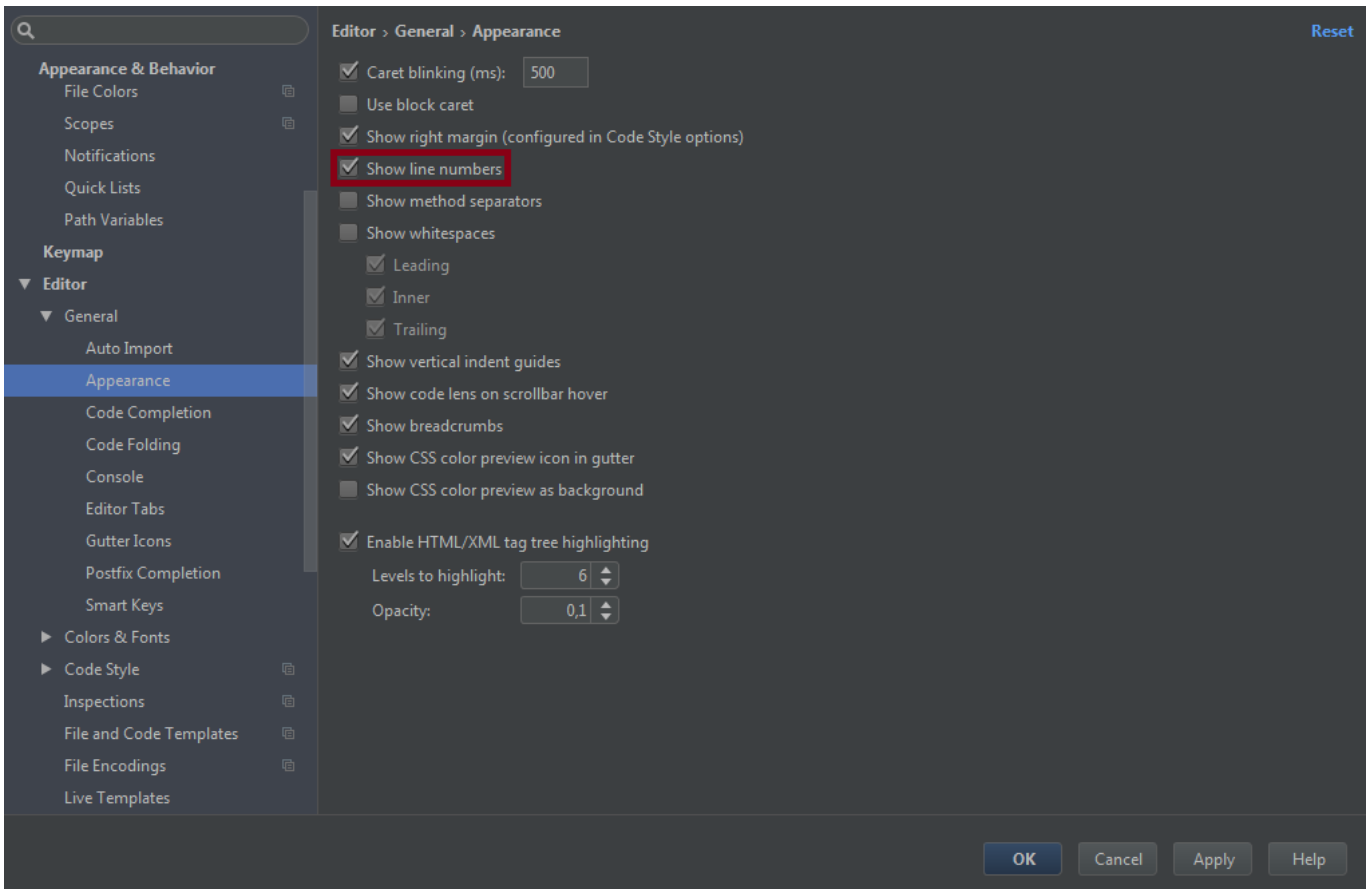
Este es el primer post de una pequeña serie sobre Android Studio. En ellos mostraremos alguna configuraciones,algún plugin, atajos de teclados y «truquillos».

- Número de linea
- Camel humps
- Imports automáticos
- Colores en el logview de Android Studio
- Plugins
    - Plugin ButterKnife Zelezny
    - Plugin Exynap
    - Markdown Navigator

Números de linea

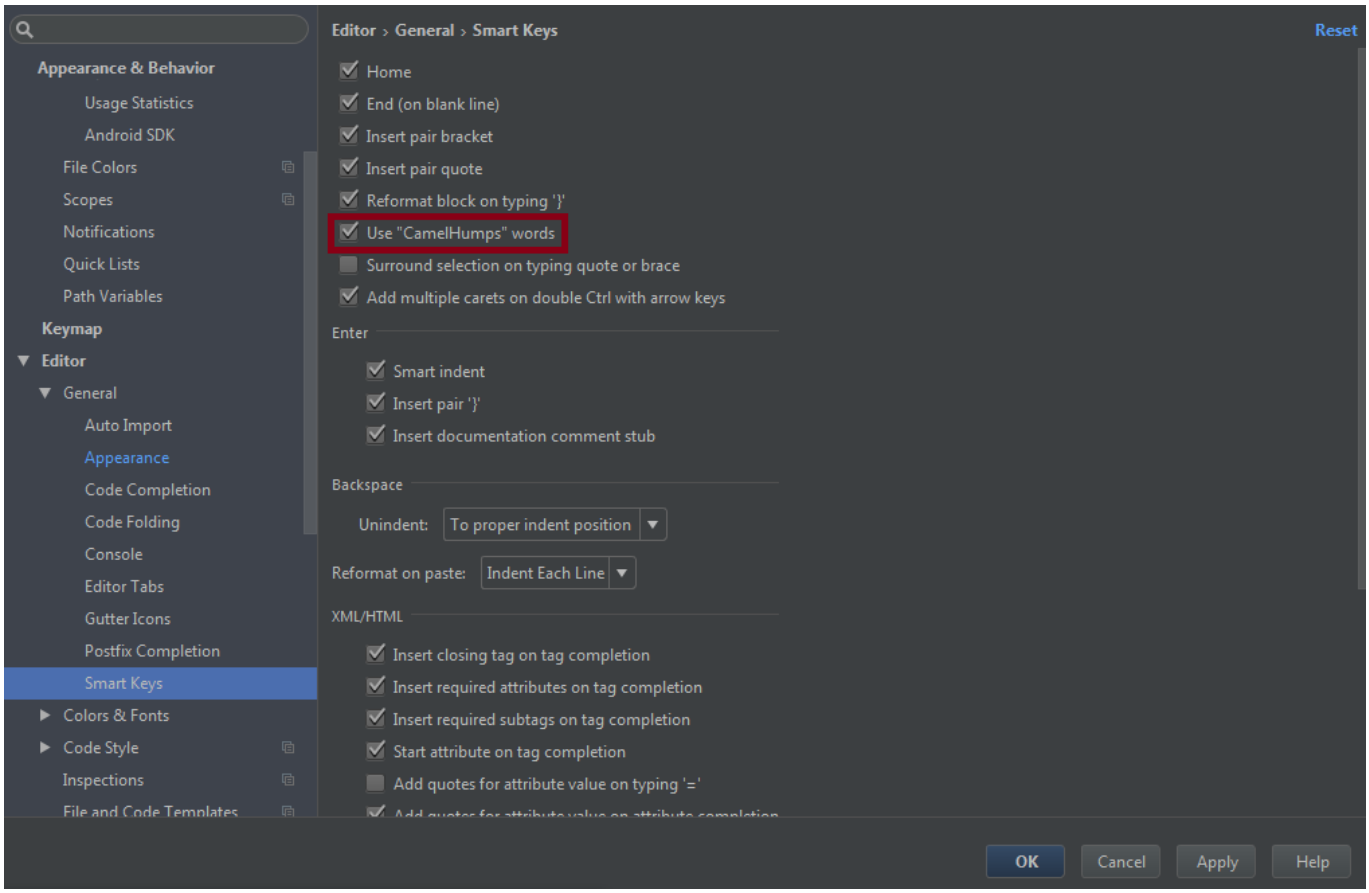
Por alguna razón, que no entiendo del todo, por defecto Android Studio no nos muestra la lineas dentro del editor.

Para habilitar esta funcionalidad deberemos acceder a ***File->Settings ->Editor->General->Appearance*** **y habilitar «*Show line numbers*».**

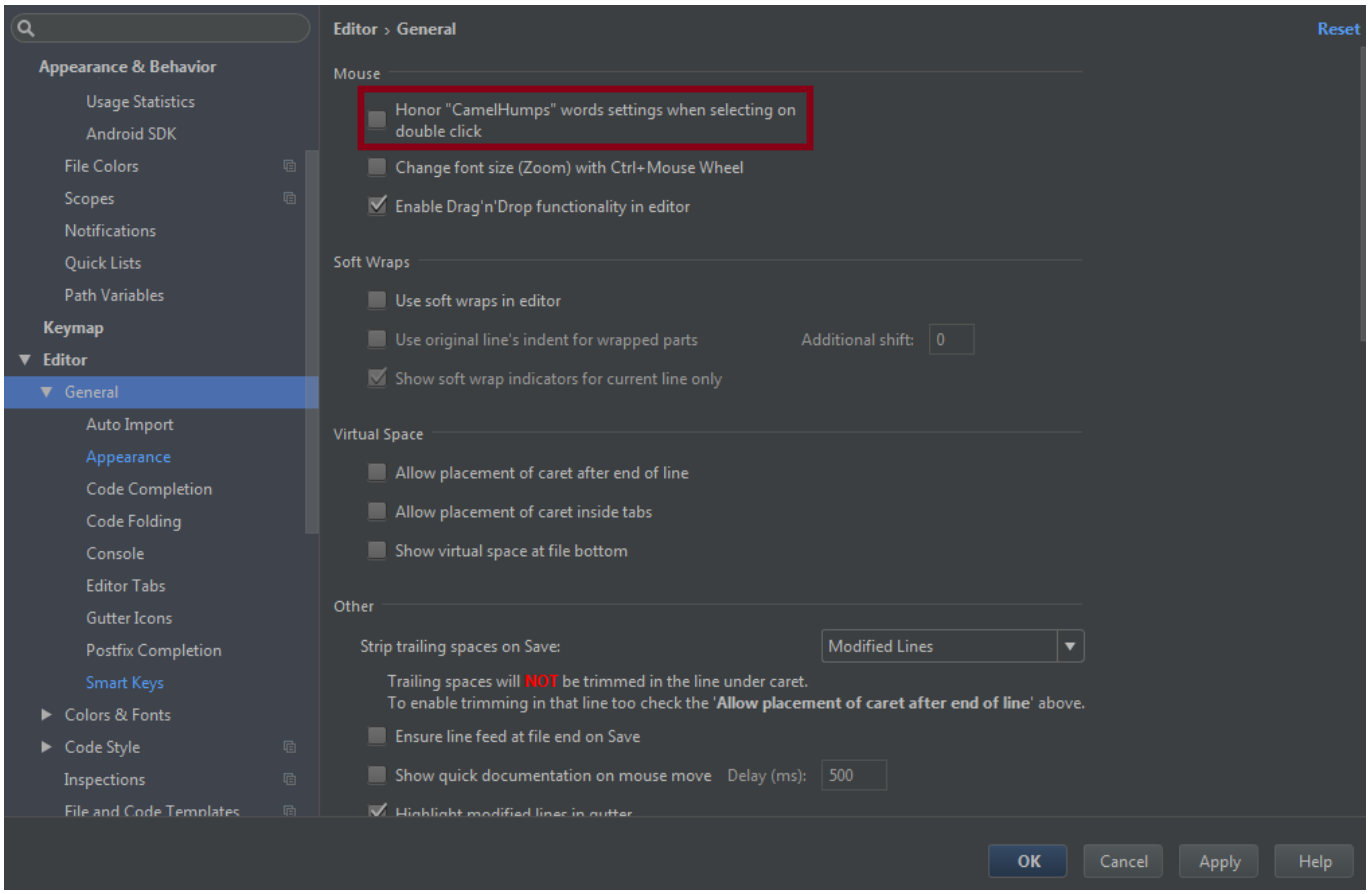


Camel humps

Android Studio no respeta las palabras ‘camel Humps’ cuando se navega a través del código con las teclas Ctrl + Izquierda / Derecha. Para solucionarlo accedemos a ***File->Settings->Editor->General->Smart Keys*** y finalmente *seleccionamos* ***Use ‘Camel Humps’ words***

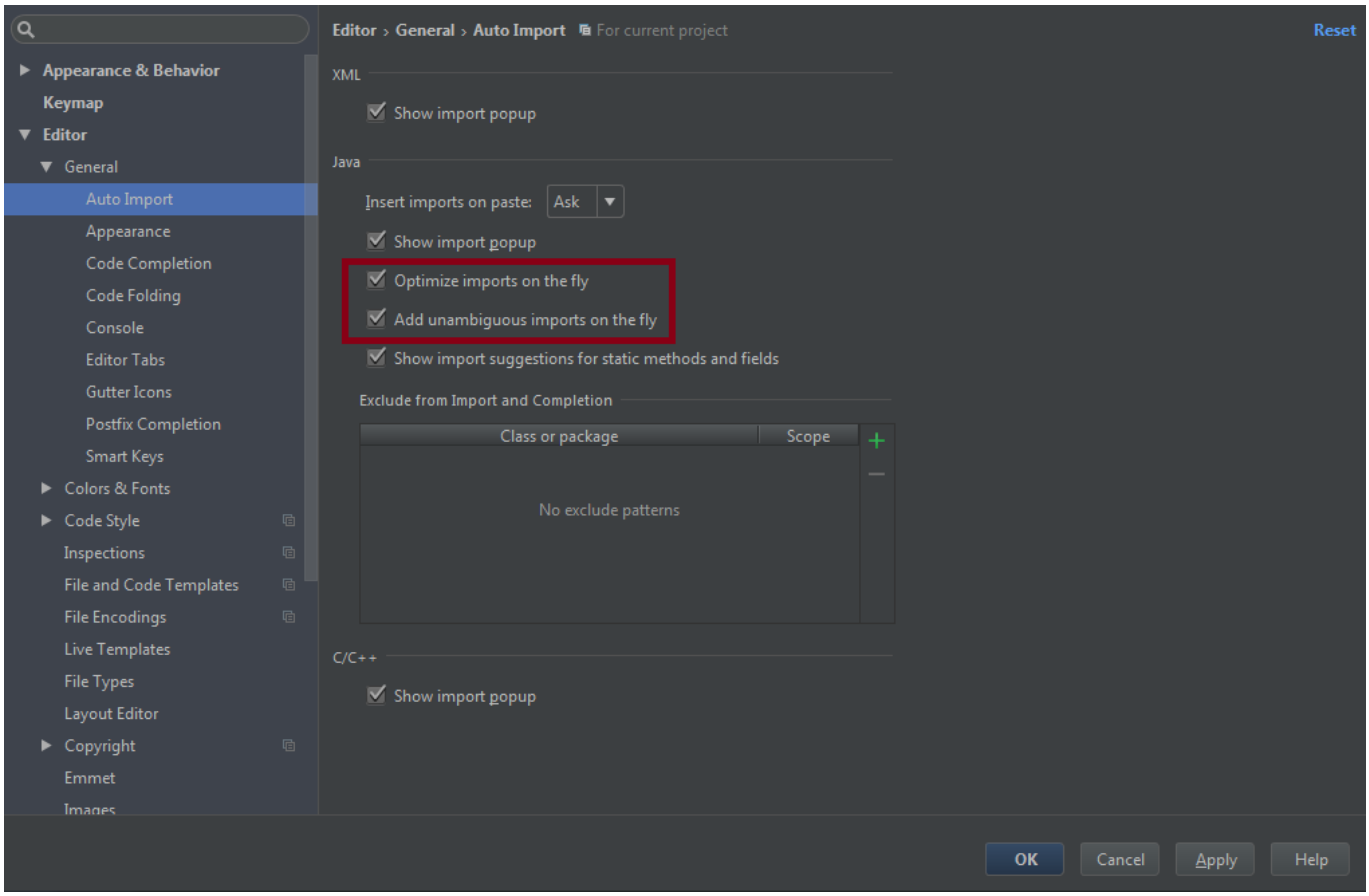


A continuación, lo que nos ocurrirá es que no podremos seleccionar con doble click una palabra. Para solucionarlo debemos acceder a ***File->Settings->Editor->General*** y deseleccionar ***Honor 'Camel Humps' words settings when selecting on double click***



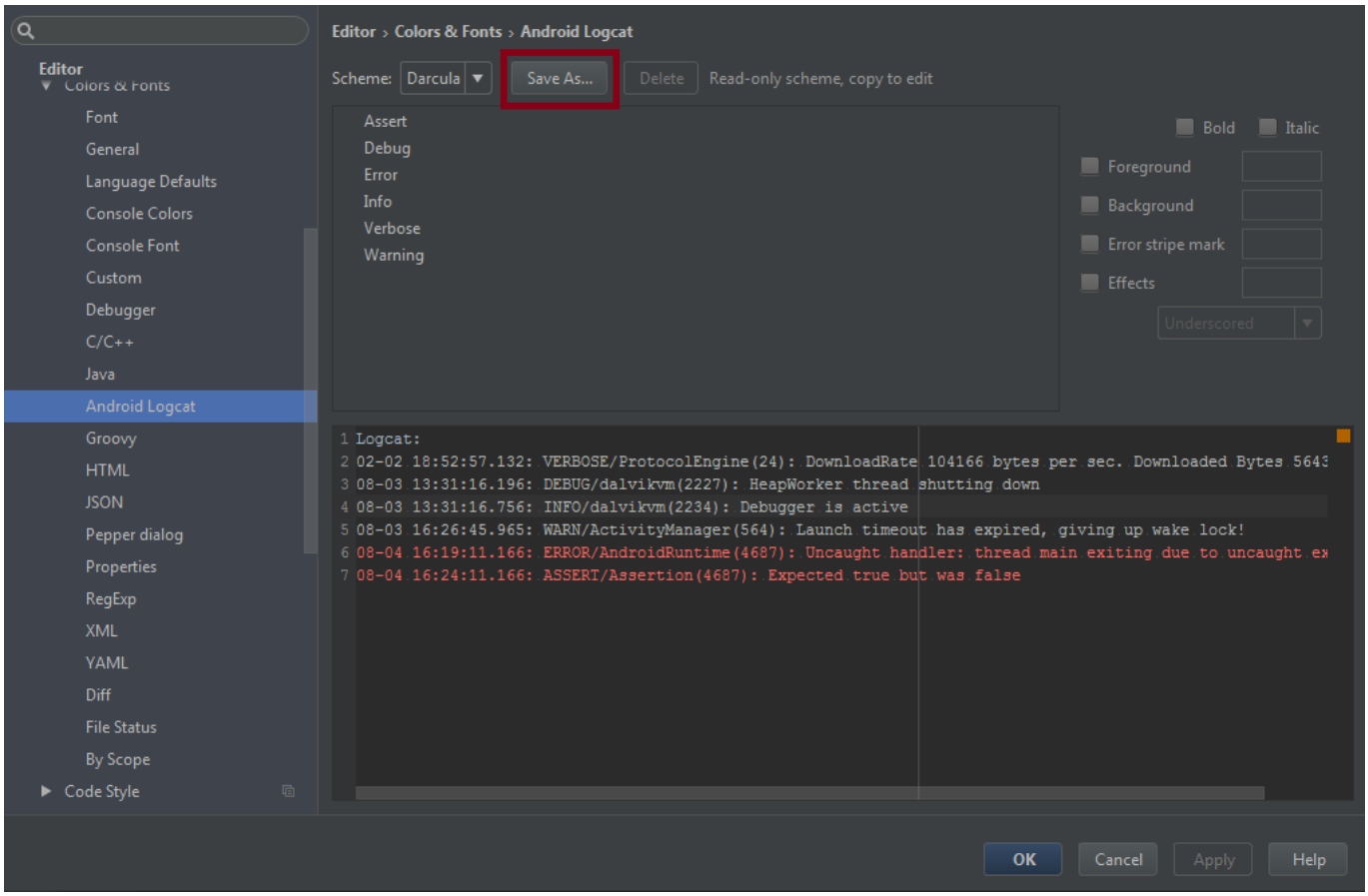
Imports Automáticos

Bien la verdad, es que hoy en día no es necesario saberse todos los packages que debemos usar, pues con una combinación de teclas podemos importarlos. Pero, ¿por que no hacerlo automáticamente? Para activarlo deberemos acceder a ***File->Settings->Editor->General->Auto Import***, seguidamente deberemos seleccionar ***Optimize imports on the fly*** y ***Add unambiguous imports on the fly***



Colores en el logview de Android Studio

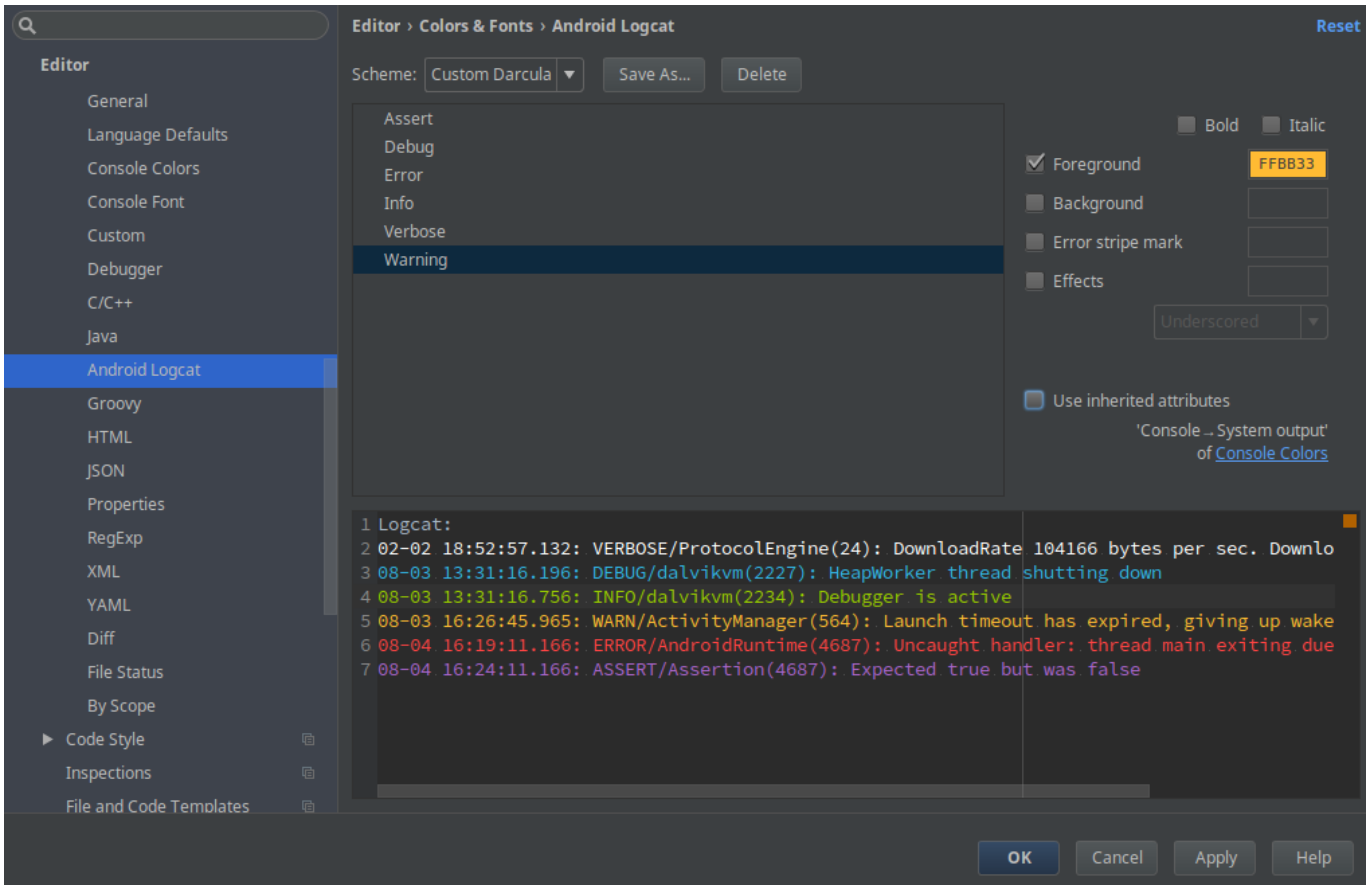
Bien esto es solo una preferencia. En logcat un buen esquema de colores nos ayudará a visualizar mejor la información. Deberemos acceder a ***File|Settings->Editor->Colors & Fonts->Android Logcat*** en este punto debemos hacer clic en ***Save As…*** y crear un nuevo esquema de colores.



Seguidamente por cada unos de los tipos de log (Assert, Debug, Error…) debemos deseleccionar ***‘Use inherited attributes’*** para cada color y aplicar el nuevo.

- Assert: #AA66CC
- Debug: #33B5E5
- Error: #FF4444
- Info: #99CC00
- Verbose: #FFFFFF
- Warning: #FFBB33

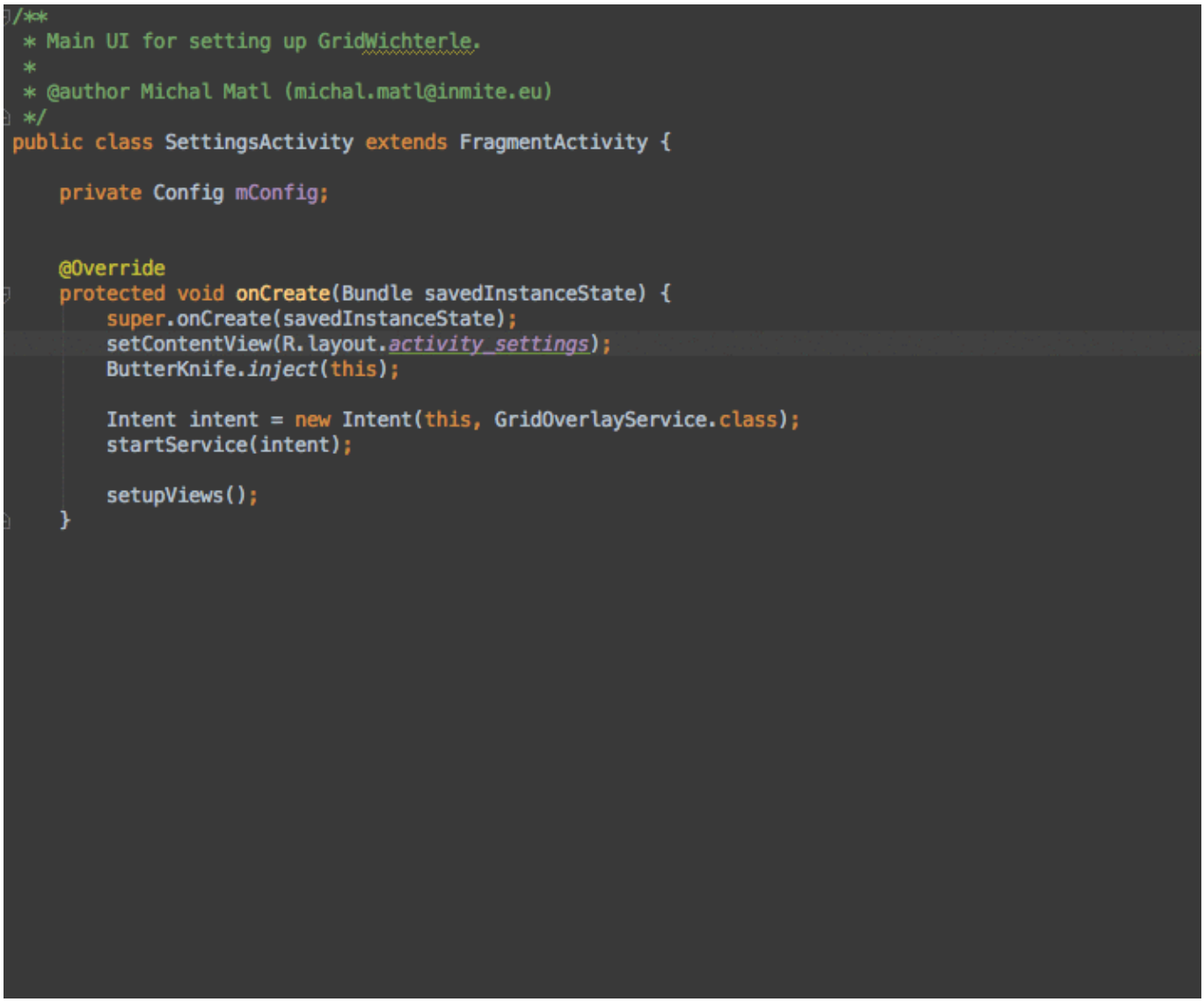
Este es el resultado



Plugins

Plugin ButterKnife Zelezny

De este plugin ya he hablado, la verdad es que nos facilita mucho el uso de Butterknife.Para instalarlo debemos acceder a ***File->settings->Plugin***, seguidamente clicar en ***Browse Repositories*** buscar ***ButterKnife Zelezny*** instalar y reiniciar. A continuación os dejo un gif de que muestra su utilización, sacado del proyecto de GitHub de este plugin. También os animo a que os paséis por la entrada de Butterknife



```
/**
 * Main UI for setting up GridWichterle.
 *
 * @author Michal Matl (michal.matl@inmite.eu)
 */
public class SettingsActivity extends FragmentActivity {

    private Config mConfig;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_settings);
        ButterKnife.inject(this);

        Intent intent = new Intent(this, GridOverlayService.class);
        startService(intent);

        setupViews();
    }
```

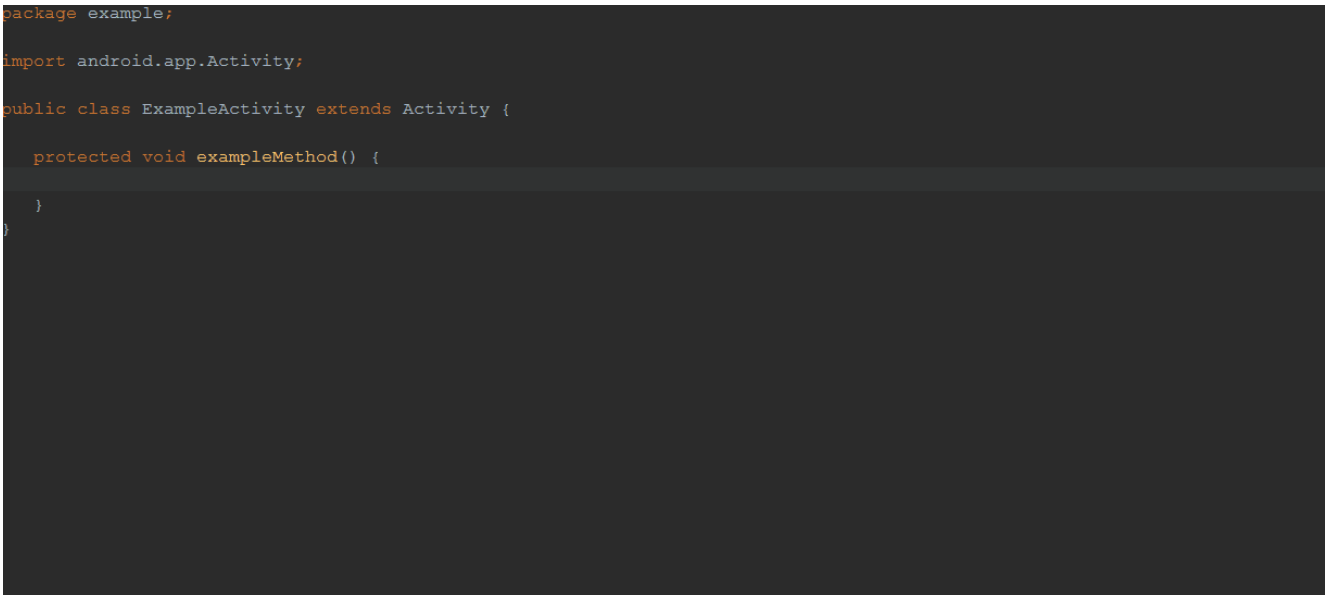
Plugin Exynap

Bien este plugin es espectacular. Nos permite encontrar e implementar el código que necesitemos en un instante. Para instalarlo debemos acceder a ***File->settings->Plugin***, seguidamente clicar en ***Browse Repositories*** buscar ***Exynap*** instalar y reiniciar.

Mediante la combinación de teclado ***Ctrl + Shift + D*** se abrirá una ventana contextual, des de la cual podremos buscar lo que necesitamos. Os dejo algunos gifs de su propia página.

```
package example;

import android.app.Activity;
import android.widget.TextView;

public class ExampleActivity extends Activity {

  protected void exampleMethod() {
    TextView textView = new TextView(this);
    textView
  }
}
```

```
package example;

import android.app.Activity;

public class ExampleActivity extends Activity {

  protected void exampleMethod() {
    int width
  }
}
```

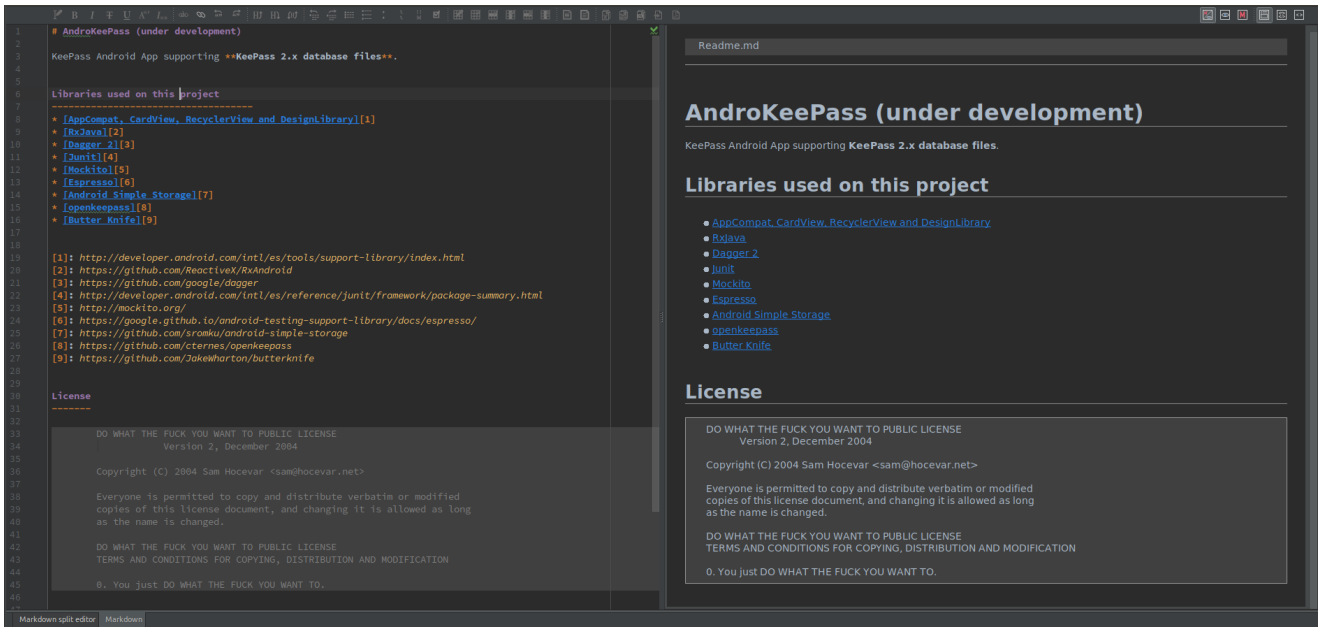


```
package example;

import android.app.Activity;

public class ExampleActivity extends Activity {

  protected void exampleMethod() {

  }
}
```

Markdown Navigator

Este plugin nos facilitara la edición de los archivos de markdown que tengamos en el proyecto. Por ejemplo ficheros de Changelog, Readme… Para instalarlo debemos acceder a ***File->settings->Plugin***, seguidamente clicar en ***Browse Repositories*** buscar ***Markdown Navigator*** instalar y reiniciar.

A continuación cuando abramos cualquier fichero de markdown se abrirá con el editor instalado.

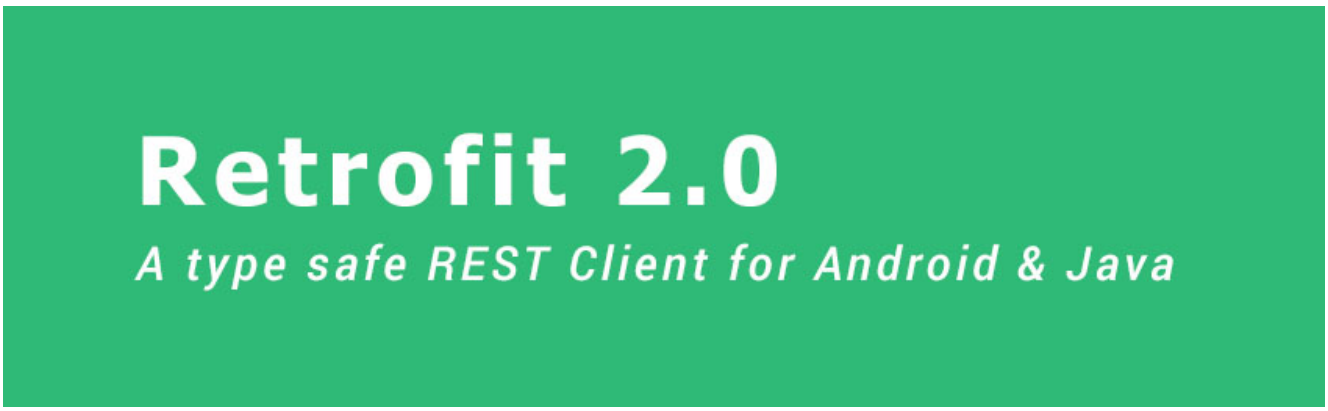


Observaciones

Bien, este post no ha sido de programación propiamente dicha. Ha sido una preconfiguración de Android Studio. La verdad es que estas pequeñas configuraciones y plugins nos ayudan a programar de manera más optima y rápida. En el siguiente post mostraremos atajos de teclado de Android Studio (que son varios y muy útiles).

Android – Retrofit y

Seeeduino Cloud



android – retrofit

Introducción

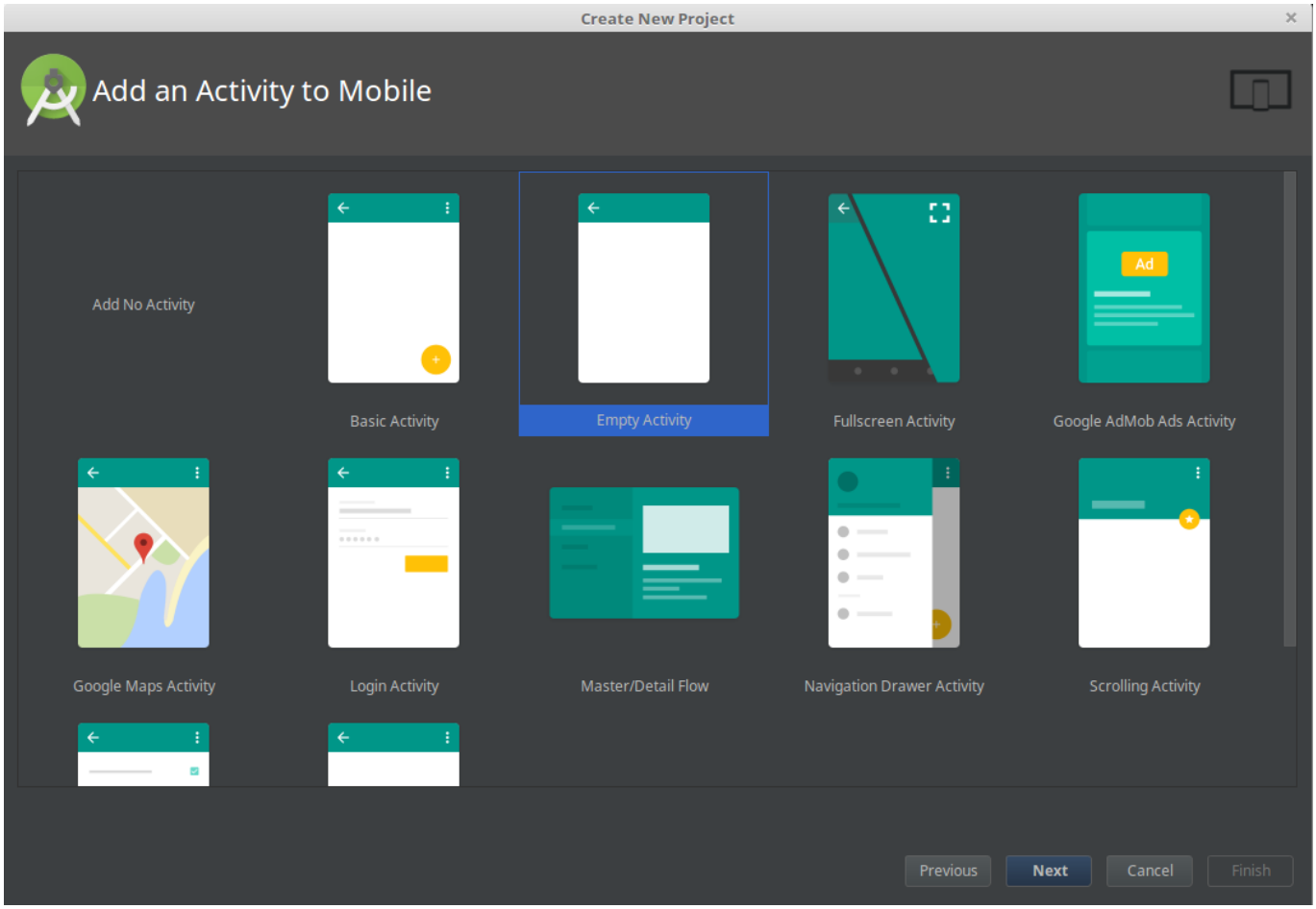
Y con este post llegamos a una de la librerías más útiles. Retrofit nos facilita el uso de llamadas rest, realizando las llamadas asíncronas sin que nosotros nos preocupemos de nada. Si además usamos GSON podemos obtener el resultado en una colección de objetos. La utilización de Retrofit en nuestros proyectos es muy fácil y ofrece un buen rendimiento. Una vez más, Square nos brinda una gran librería (ya hemos hablado anteriormente de Picasso).

Este post es una continuación de los de seeeduino (Seeduino Cloud – Parte 1 , Seeeduino Cloud – Parte 2). Al finalizarlo podremos controlar nuestro ventilador des de una aplicación Android.

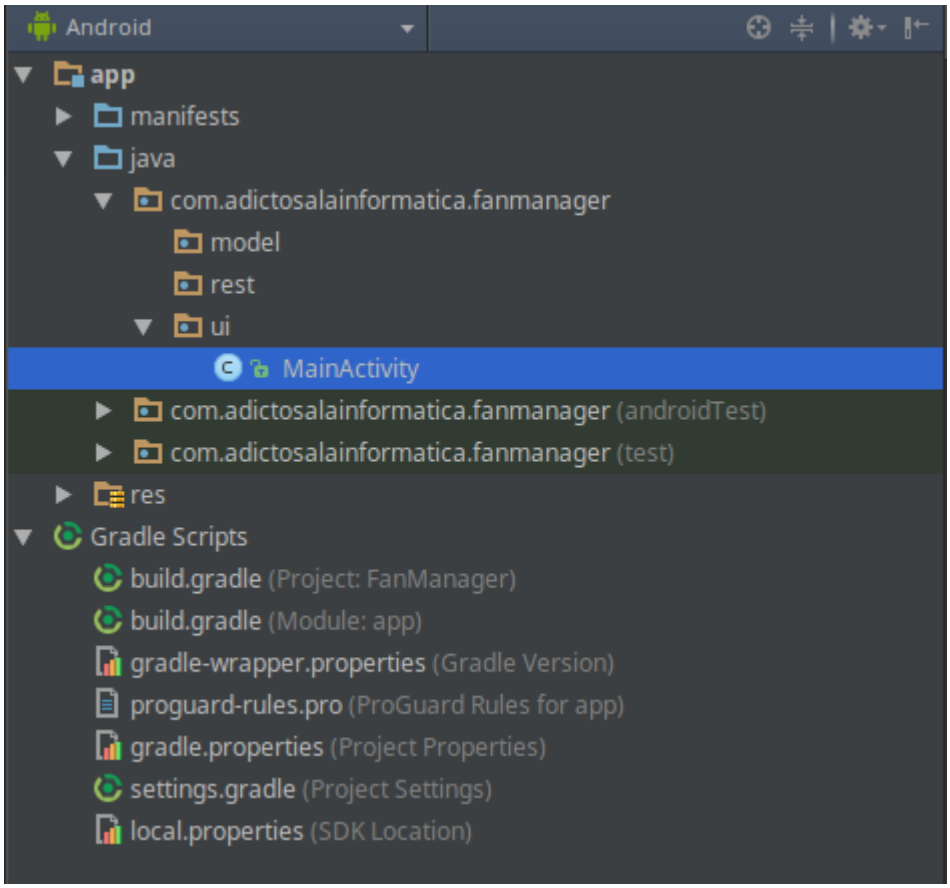
- Creando la estructura del proyecto
- Configuración
- Creando nuestras clases Pojo
- Creando una instancia de Retrofit
- Preparando end points
- Preparando nuestro activity

Creando el proyecto y su estructura

Crearemos un proyecto con un empty activity



A continuación crearemos una nueva estructura de packages. Esto es una costumbre, no es ningún estándar y cada developer utiliza la que más le conviene o resulta más fácil de usar, para organizar su código. Muy probablemente, cambiará durante el proceso desarrollo y aquí cada unos tiene sus preferencias. Lo que realmente es importante es utilizar una que nos ayude mantener nuestro código debidamente organizado.



Configuración

Añadiremos el plugin android-apt a nuestro classpath en el fichero **build.gradle**. Este se encuentra en la raíz de nuestro proyecto.

```
dependencies{
  classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
}
```

Dentro del archivo **app/build.gradle**, debemos añadir las dependencias de Retrofit.

```
apply plugin : 'com.neenbedankt.android-apt'

dependencies {

  // retrofit, gson
  compile 'com.google.code.gson:gson:2.6.2'
  compile 'com.squareup.retrofit2:retrofit:2.0.2'
  compile 'com.squareup.retrofit2:converter-gson:2.0.2'
```

```
  // picasso
  compile 'com.squareup.picasso:picasso:2.5.2'
  compile 'jp.wasabeef:picasso-transformations:2.1.0'

  // butterknife
  compile 'com.jakewharton:butterknife:8.4.0'
  apt 'com.jakewharton:butterknife-compiler:8.4.0'
}
```

Seguidamente añadiremos el permiso de red y de comprobación de esta a nuestro archivo **androidmanifest.xml**

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE"></uses-
permission>
```

Creando nuestra clase Pojo

Esta es la clase base sobre la que Gson creará la instancia con los resultados de la llamada rest realizada con Retrofit, colocaremos la clase en el package **model**. Una de las maneras más fáciles de generar nuestras clases es utilizar un generador. Por ejemplo, jsonschema2pojo. Pero debemos tener cuidado, podemos encontrarnos fácilmente con una respuesta Json inmensa de la cual tan solo necesitamos alguna información concreta.

```
package com.adictosalainformatica.fanmanager.model;

import com.google.gson.annotations.Expose;
import com.google.gson.annotations.SerializedName;

public class FanModel {

    @SerializedName("pin")
    @Expose
    private Integer pin;
    @SerializedName("status")
    @Expose
```

```
    private Integer status;

    /**
     *
     * @return
     * The pin
     */
    public Integer getPin() {
        return pin;
    }

    /**
     *
     * @param pin
     * The pin
     */
    public void setPin(Integer pin) {
        this.pin = pin;
    }

    /**
     *
     * @return
     * The status
     */
    public Integer getStatus() {
        return status;
    }

    /**
     *
     * @param status
     * The status
     */
    public void setStatus(Integer status) {
        this.status = status;
    }

}
```

Es importante no acabar con una colección de clases inmensa

llena de getters y setters que no vamos a utilizar. Todo ello para mantener nuestro código limpio o no acabar encontrándonos el error «64k method limit in dex». No es raro encontrarnos con él si utilizamos muchas librerías y colecciones de clases generadas a partir del resultado de una respuesta Json desmesurada. Podemos fácilmente solventar el problema revisando nuestras clases Pojo y eliminando todas aquellas variables y clases que no vamos a utilizar e incluso plantearnos si todas las librerías que estamos utilizando son realmente necesarias. Pero si esto no fuera suficiente, podemos solventar el problema con multidex a costa de perder la funcionalidad de instant run.

Creando una instancia de Retrofit

Para enviar solicitudes de red a una API, tenemos que utilizar la clase Retrofit.Builder y especificar la URL base para el servicio. Por lo tanto, crearemos una clase llamada ApiClient.java bajo en el package **rest**.

BASE_URL – es la url base de nuestra API. Vamos a utilizar esta url para todas las solicitudes posteriores.

```
package com.adictosalainformatica.fanmanager.rest;

import retrofit2.Retrofit;
import retrofit2.converter.gson.GsonConverterFactory;


public class ApiClient {

          public static final String BASE_URL =
"http://192.168.1.33/arduino/";
    private static Retrofit retrofit = null;


    public static Retrofit getClient() {
        if (retrofit==null) {
```

```
            retrofit = new Retrofit.Builder()
                    .baseUrl(BASE_URL)
.addConverterFactory(GsonConverterFactory.create())
                    .build();
        }
        return retrofit;
    }
}
```

Preparando end points

Los end points se definen dentro de una interfaz mediante anotaciones especiales de Retrofit para codificar información sobre los parámetros y el tipo de petición. Además, el valor de retorno es siempre una llamada con parámetros <T>, en nuestro caso <FanModel>. Crearemos la interface ApiInterface.java en el package **rest**

```
package com.adictosalainformatica.fanmanager.rest;

import
com.emotionexperience.fragancemanager.model.FraganceModel;

import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Path;


public interface ApiInterface {
    @GET("digital/{pin}/{value}")
     Call setPin(@Path("pin") int pin, @Path("value") int
value);

    @GET("status/{pin}")
    Call getStatus(@Path("pin") int pin);
}
```

Preparando nuestro activity

A continuación mostramos el código de nuestro activity que se encuentra en el package **ui**. Como se puede observar utilizamos Picasso y Butterknife

```
package com.adictosalainformatica.fanmanager.ui;

import android.app.ProgressDialog;
import android.content.Context;
import android.graphics.Color;
import android.net.ConnectivityManager;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.widget.ImageButton;
import android.widget.Toast;

import com.adictosalainformatica.fanmanager.R;
import com.adictosalainformatica.fanmanager.model.FanModel;
import com.adictosalainformatica.fanmanager.rest.ApiClient;
import com.adictosalainformatica.fanmanager.rest.ApiInterface;
import com.squareup.picasso.Picasso;

import butterknife.BindView;
import butterknife.ButterKnife;
import butterknife.OnClick;
import
jp.wasabeef.picasso.transformations.ColorFilterTransformation;
import
jp.wasabeef.picasso.transformations.CropCircleTransformation;
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.Response;

public class MainActivity extends AppCompatActivity {

    @BindView(R.id.main_btn_fan)
    ImageButton btnFan;

    // Constants
```

```
            private static final String TAG =
MainActivity.class.getName();
    private static int PIN = 8;

    private int fanStatus = 0;
    private static ApiInterface apiService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ButterKnife.bind(this);

        Picasso
                .with(getApplicationContext())
                .load(R.drawable.fan_image)
                .transform(new CropCircleTransformation())

                .into(btnFan);

                                          apiService     =
ApiClient.getClient().create(ApiInterface.class);

        if(isConetctionEnabled(getApplicationContext())){
            getFanStatus(PIN);
        }else{
             Toast.makeText(getApplicationContext(),"Internet
connection failed",Toast.LENGTH_LONG).show();
        }
    }

    @OnClick(R.id.main_btn_fan)
    public void switchFanStatus() {
        if(isConetctionEnabled(getApplicationContext())){
            if(fanStatus == 0){
                setFanStatus(PIN,1);
            }else{
                setFanStatus(PIN, 0);
            }
        }else{
```

```
            Toast.makeText(getApplicationContext(),"Internet
connection failed",Toast.LENGTH_LONG).show();
        }
    }

    /**
     * Get current Fan status
     * @param pin
     */
    private void getFanStatus(int pin) {
        Call <FanModel> call = apiService.getStatus(pin);
        call.enqueue(new Callback<FanModel>() {
            @Override
              public void onResponse(Call<FanModel> call,
Response<FanModel> response) {
                if (response.isSuccessful()) {
                    fanStatus =  response.body().getStatus();
                    Log.d(TAG, "Fan status: " + fanStatus);

                    if(fanStatus == 0){
                                              int  color  =
Color.parseColor("#33ee092b");
                        setColorFilter(color);

                    }else{
                                              int  color  =
Color.parseColor("#3300ff80");
                        setColorFilter(color);
                    }
                } else {
                    //request not successful (like 400,401,403
etc)
                    Log.e(TAG,response.message());
                }
            }

            @Override
               public void onFailure(Call<FanModel> call,
Throwable t) {
                // Log error here since request failed
                Log.e(TAG, "Error: " + t.toString());
```

```
            }
        });
    }

    /**
     * Set Fan status
     * @param pin
     */
    private void setFanStatus(int pin, int status){
        Call call = apiService.setPin(pin,status);
        call.enqueue(new Callback<FanModel>() {
            @Override
               public void onResponse(Call<FanModel> call,
Response<FanModel> response) {
                if (response.isSuccessful()) {
                    fanStatus =  response.body().getStatus();
                    Log.d(TAG, "Fan status: " + fanStatus);

                    if(fanStatus == 0){
                                              int  color  =
Color.parseColor("#33ee092b");
                        setColorFilter(color);

                    }else{
                                              int  color  =
Color.parseColor("#3300ff80");
                        setColorFilter(color);
                    }
                } else {
                    //request not successful (like 400,401,403
etc);
                    Log.e(TAG,response.message());
                }
            }

            @Override
               public void onFailure(Call<FanModel> call,
Throwable t) {
                // Log error here since request failed
                Log.e(TAG, "Error: " + t.toString());
            }
```

```
        });
    }

    /**
     * Set image filter color
     * @param color
     */
    private void setColorFilter(int color){
        Picasso
                .with(getApplicationContext())
                .load(R.drawable.fan_image)
                                                .transform(new
ColorFilterTransformation(color))
                .transform(new CropCircleTransformation())
                .into(btnFan);
    }

    /**
     * Tests if there's connection
     * @param cx context application
     * @return true or false
     */
    public static boolean isConetctionEnabled(Context cx){
                        ConnectivityManager   conMgr   =
(ConnectivityManager)cx.getSystemService(Context.CONNECTIVITY_
SERVICE);

        if (conMgr.getActiveNetworkInfo() != null
                && conMgr.getActiveNetworkInfo().isAvailable()
                                                            &&
conMgr.getActiveNetworkInfo().isConnected()) {
            return true;
        } else {
            return false;
        }
    }
}
```

I finalmente, nuestro layout

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
tools:context="com.adictosalainformatica.fanmanager.ui.MainAct
ivity">


    <ImageButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:srcCompat="@drawable/fan_image"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true"
        android:id="@+id/main_btn_fan"
        android:background="@null"
        android:padding="10dp"/>
</RelativeLayout>
```

Desde este enlace os podéis descargar fan_image

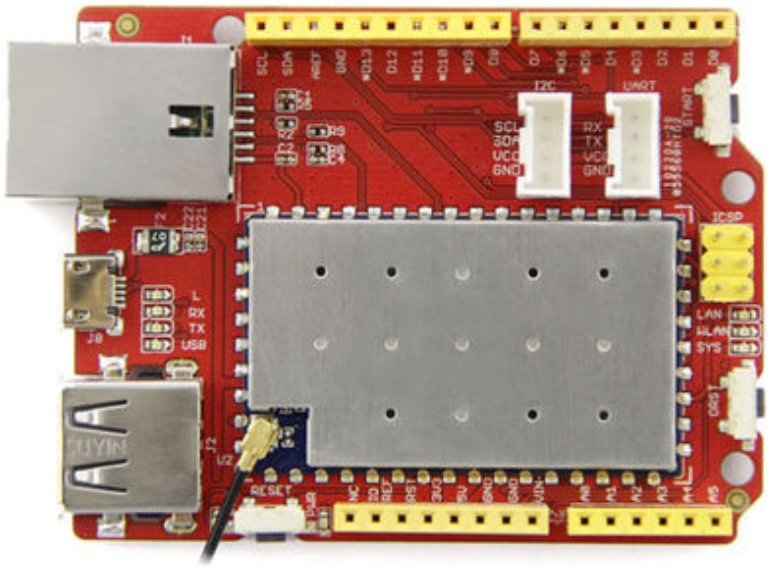
Observaciones

Después de mucho tiempo lidiando con AsyncTask, rotaciones, memory leaks… Retrofit nos permite abstraernos de todo esto y además es muy fácil de utilizar. Por todo ello, Retrofit se convierte en una librería casi indispensable. Finalmente, dejo el enlace a la web de Retrofit y un ejemplo en Github. El cual, nos permite apagar y encender un ventilador siempre y cuando tengamos nuestro Seeduino Cloud configurado y preparado para trabajar con un relayshield.

- Retrofit
- Seeeduino Cloud – Parte 1 (configuración Seeduino Cloud)
- Seeeduino Cloud – Parte 2 (Montaje relayshield)

- FanManager

Seeeduino Cloud – Parte 2



Seeeduino Cloud

Introducción

En este post añadiremos una relay shield a nuestro seeeduino. Seguidamente, realizaremos el cableado con uno de los relay y conectaremos un pequeño ventilador. De esta manera podremos encender y apagar nuestro ventilador remotamente.

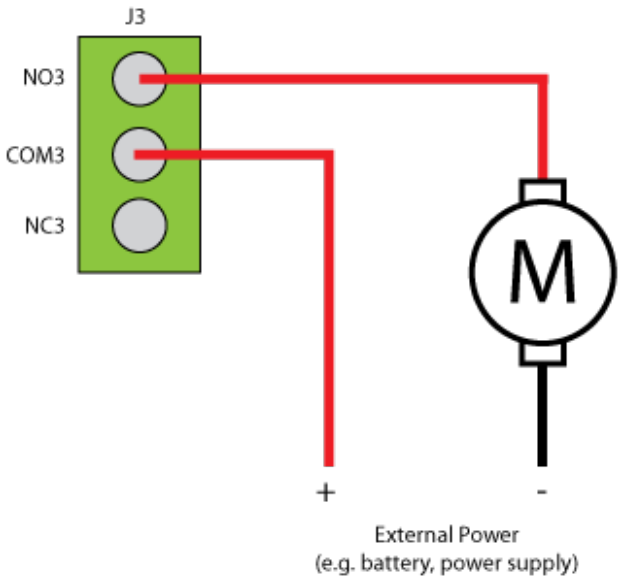
- Montando el Relay Shield y cableado de un relay
- Programa para controlar el relay
- Encendiendo y apagando nuestro ventilador

Montando el Relay Shield y cableando un relay

El funcionamiento de un relay es bastante simple. Un relay es un switch electromagnético. Un relay tiene tres entradas:

- **nc** (normally closed). El circuito tiene corriente, a no ser que se active el relay.
- **no** (normally open). El circuito no tiene corriente, a no ser que se active el relay.
- **com** (entrada de electricidad). Punto de entrada del corriente eléctrico externo.

En el siguiente ejemplo podemos ver la conexión de un relay con un motor. Si observamos el diagrama el motor no estará en funcionamiento hasta que se active el relay.

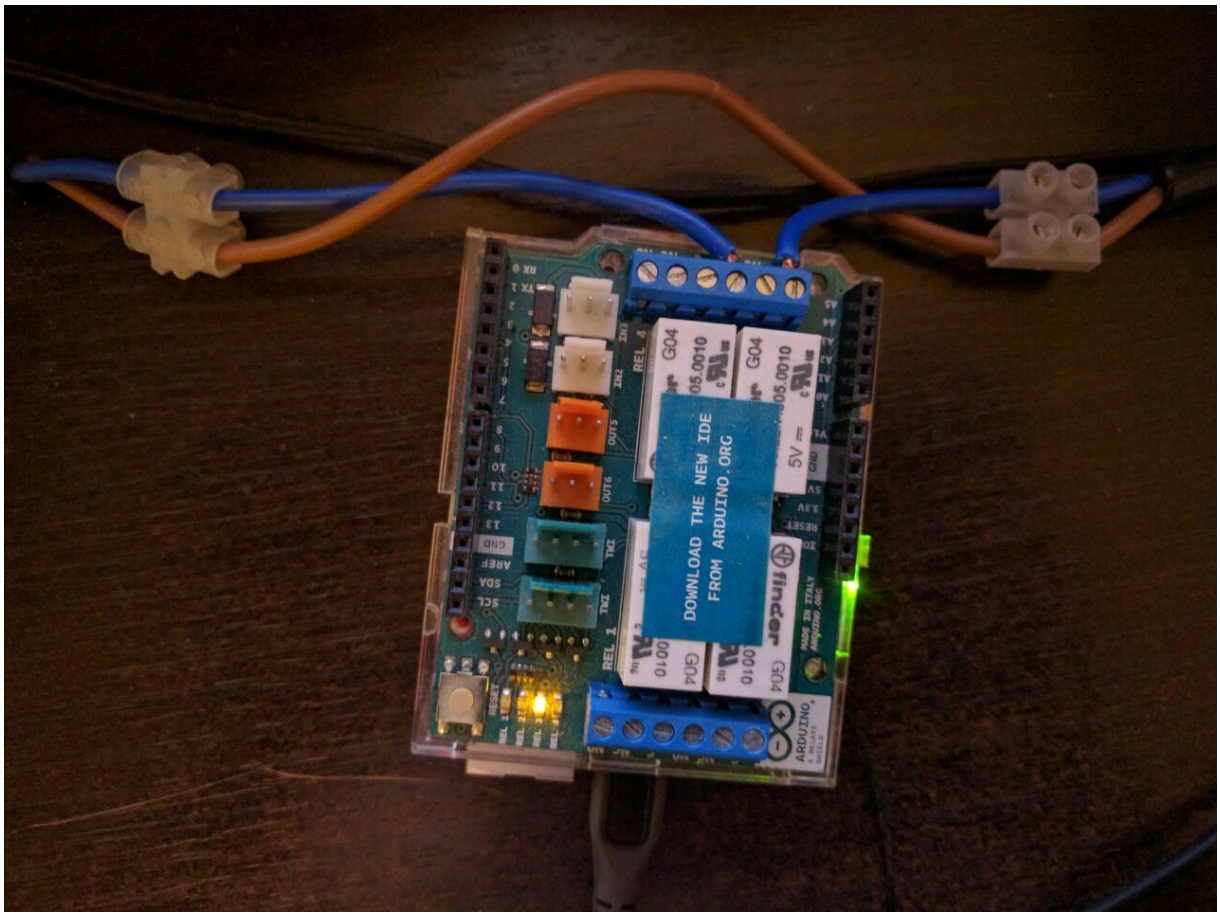


Motor-shield-schematic-drawing

Primeramente montaremos la relay shield encima de nuestro Seeeduino (Encaja perfectamente).

Seguidamente cablearemos un relay. En este ejemplo utilizaremos el relay 3 (digital pin 8), pero se podría utilizar cualquiera. La entrada de corriente estará en el conector **C**

, y la salida en el conector **NO**



Seguidamente la embra:

Y finalmente la toma de electricidad:

Solo nos quedará conectar el macho a la corriente y un ventilador a la hembra.

Para más información sobre el relay shield utilizado os dejo el link de la página del mismo Arduino – 4 Relays Shield

Programa para controlar el relay

A continuación este es el programa que permitirá acceder remotamente a nuestro relay. El código esta debidamente comentado

```
#include <Bridge.h>
```

```
#include <BridgeServer.h>
#include <BridgeClient.h>

// Listen to the default port 5555, the Yún webserver
// will forward there all the HTTP requests you send
BridgeServer server;

int RELAY3 = 8;

void setup() {
  // Bridge startup
  pinMode(13, OUTPUT);
  digitalWrite(13, LOW);
  Bridge.begin();
  digitalWrite(13, HIGH);
  pinMode(RELAY3, OUTPUT);

  // Listen for incoming connection only from localhost
  // (no one from the external network could connect)
  server.listenOnLocalhost();
  server.begin();
}

void loop() {
  // Get clients coming from server
  BridgeClient client = server.accept();

  // There is a new client?
  if (client) {
    // Process request
    process(client);

    // Close connection and free resources.
    client.stop();
  }

  delay(50); // Poll every 50ms
}

void process(BridgeClient client) {
  // read the command
```

```
  String command = client.readStringUntil('/');

  // is "digital" command?
  if (command == "digital") {
    digitalCommand(client);
  }

  // is "status" command?
  if (command == "status") {
    statusCommand(client);
  }
}

void digitalCommand(BridgeClient client) {
  int pin, value;

  // Read pin number
  pin = client.parseInt();

  // If the next character is a '/' it means we have an URL
  // with a value like: "/digital/13/1"
  if (client.read() == '/') {
    value = client.parseInt();
    digitalWrite(pin, value);
  } else {
    value = digitalRead(pin);
  }

  // Send feedback to client
  client.print(F("{\"pin\":"));
  client.print(pin);
  client.print(F(" ,\"status\":"));
  client.print(value);
  client.print(F("}"));

  // Update datastore key with the current pin value
  String key = "D";
  key += pin;
  Bridge.put(key, String(value));

}
```

```
void statusCommand(BridgeClient client) {
  int pin, value;

  // Read pin number
  pin = client.parseInt();
  value = digitalRead(pin);

  // Send feedback to client
  client.print(F("{\"pin\":"));
  client.print(pin);
  client.print(F(" ,\"status\":"));
  client.print(value);
  client.print(F("}"));
}
```

Encendiendo y apagando nuestro ventilador

Desde un navegador accediendo a la url obtendremos el estado del relay:

http://ip-seeduino/arduino/status/8

{"pin":8 ,"status":0}

Retornará un 0 (apagado) o 1 (encendido)

Para apagarlo

http://ip-seeeduino/arduino/digital/8/0

{"pin":8 ,"status":0}

Y para encenderlo

http://ip-seeeduino/arduino/digital/8/1

{"pin":8 ,"status":1}

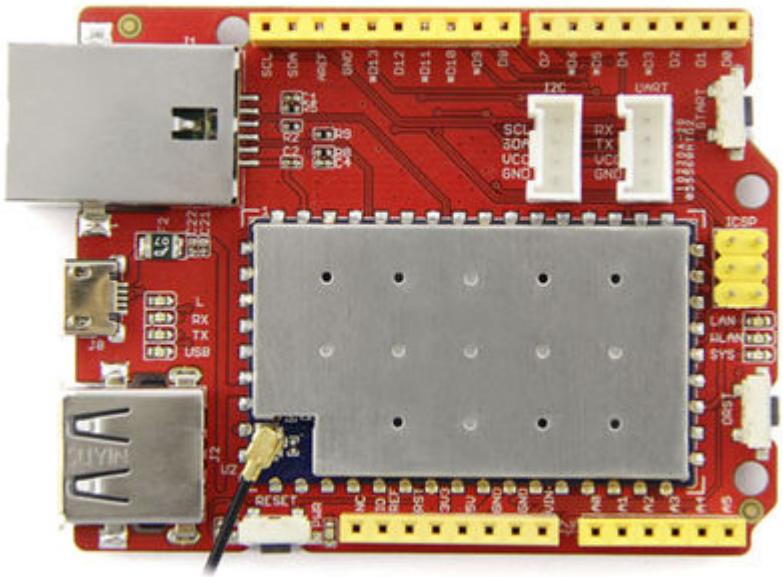
Conclusión

Ya podemos controlar nuestro ventilador remotamente y como

podemos ver ha resultado muy fácil. Pero la verdad, con una url desde el plugin de un navegador es muy rudimentario y no excesivamente útil. En el próximo post crearemos una simple aplicación Android, que se encargará de controlar y obtener el estado de nuestro ventilador des de la Rest Api de Seeeduino. De paso presentaremos y utilizaremos una librería muy útil para este tipo de escenarios, Retrofit. A continucación podéis encontrar el ejemplo del código arduino en GitHub y el enlace a la placa de rayls de arduino.

- Arduino 4 Relay Shield
- Seeeduino Rest Api example

Seeeduino Cloud – Parte 1



Introducción

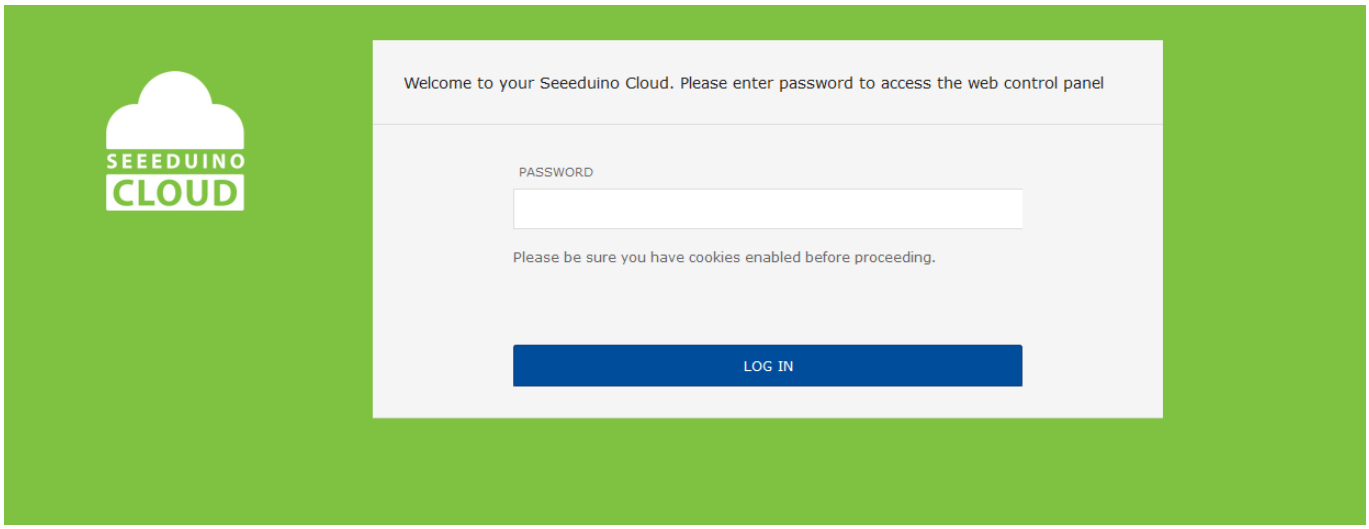
Recientemente he adquirido un Seeeduino Cloud. Un clon

compatible con Yun. La verdad es que es muy versátil, fácil de configurar y potente. En este primer post hablaremos un poco de él, lo configuraremos y habilitaremos su Api Rest para encender y apagar el led que incorpora la placa.

- Configuración de red
- Configuración con WebGui
- Programa de ejemplo y test

Configuración de red

Cuando arranquemos por primera vez nuestro Seeeduino, este levantará una wifi llamada **SeeeduinoCloud-AXXXX** a la que nos podremos conectar. Una vez echo esto, podremos acceder a la configuración web a través de la ip **192.168.240.1.**



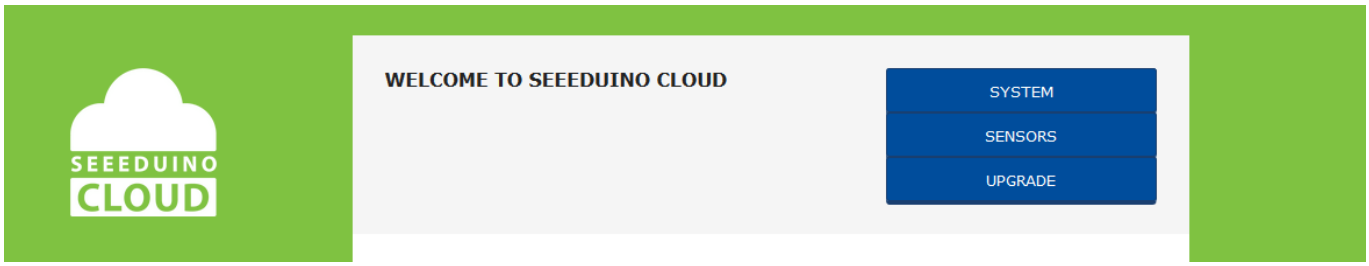
El password por defecto es **seeeduino**.

Configuración con WebGui

Después de entrar en la web, la interfaz nos mostrará el estatus de las redes WiFi/ Eth. En la parte superior derecha encontraremos las siguientes opciones:

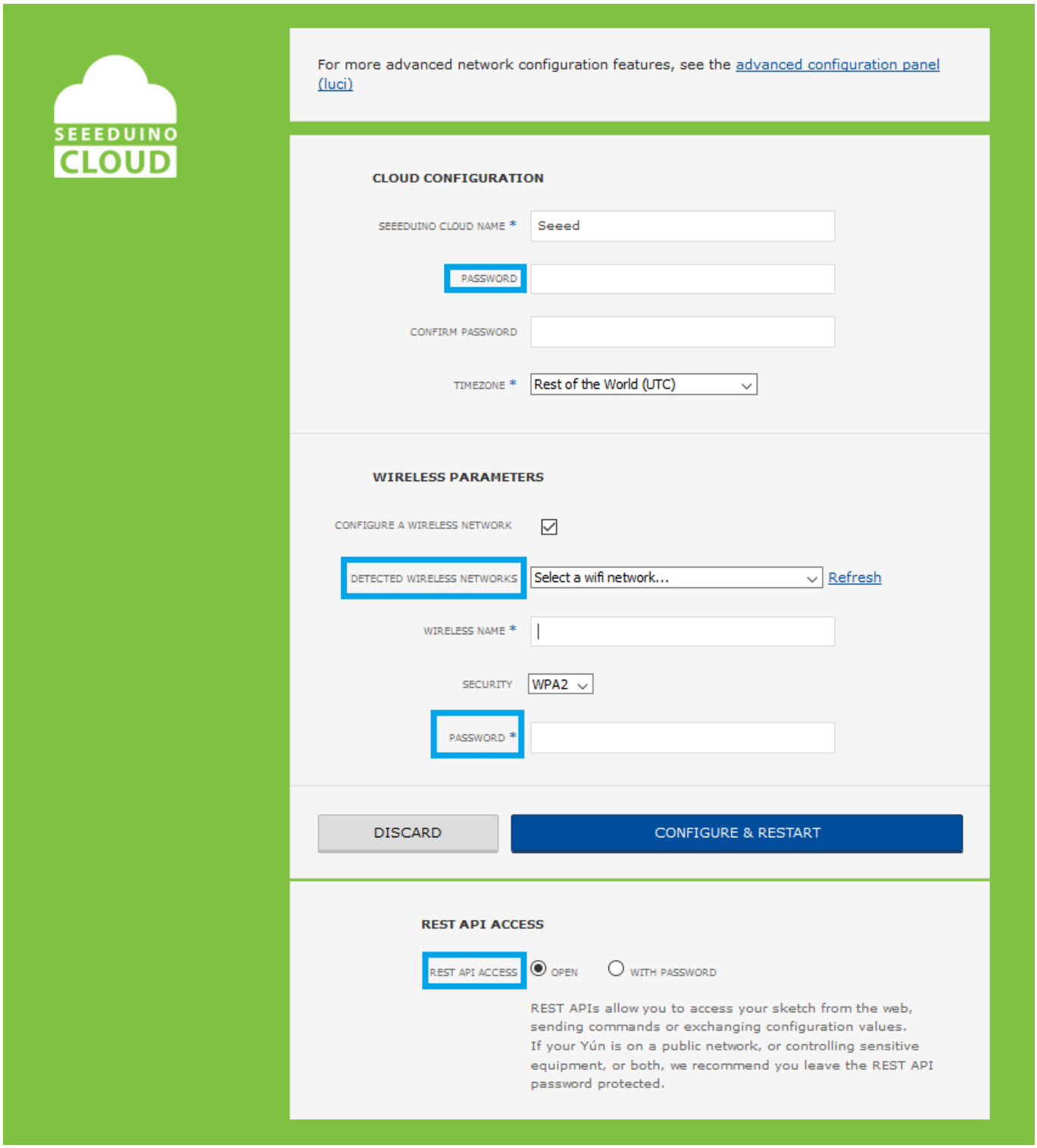
- SYSTEM -> Configuración global
- SENSORS -> configuración del servidor IoT

- UPGRADE -> Actualización de firmware



Seleccionaremos SYSTEM y configuraremos nuestro Seeeduino Cloud:

- Primeramente podemos modificar el password de acceso
- A continuación seleccionaremos la red wifi a la que queremos conectar nuestro Seeeduino Cloud y estableceremos el password de la misma
- Finalmente, podemos proteger nuestra red con una password para la Api Rest. Si lo hacemos debemos tener en cuenta que será una protección Basic Atuh y el usuario por defecto será root

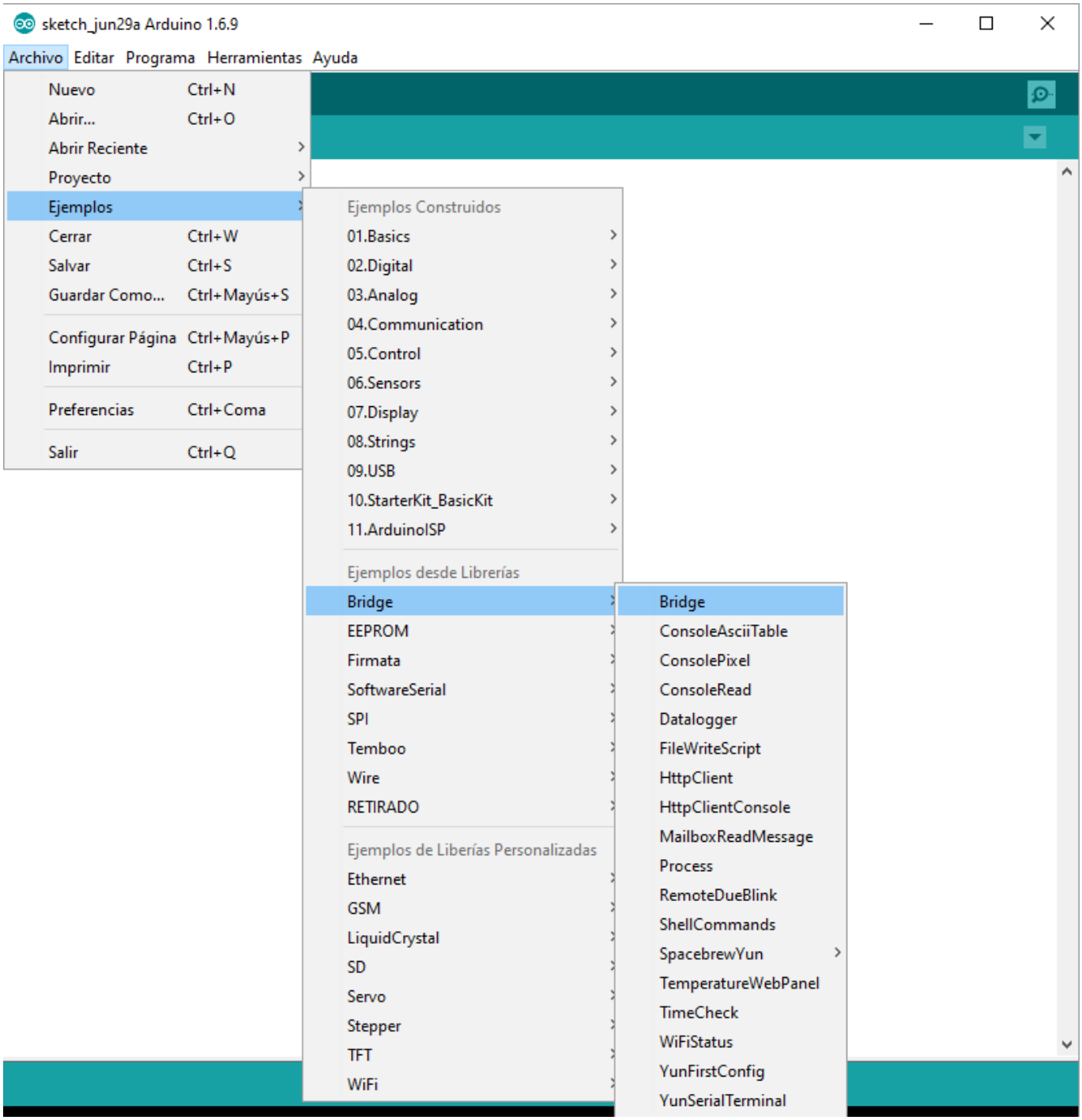


Cuando pulsemos en **CONFIGURE & RESTART** nuestro Seeeduino se configurará, se reiniciará y se conectará a nuestra red.

Programa de ejemplo y test

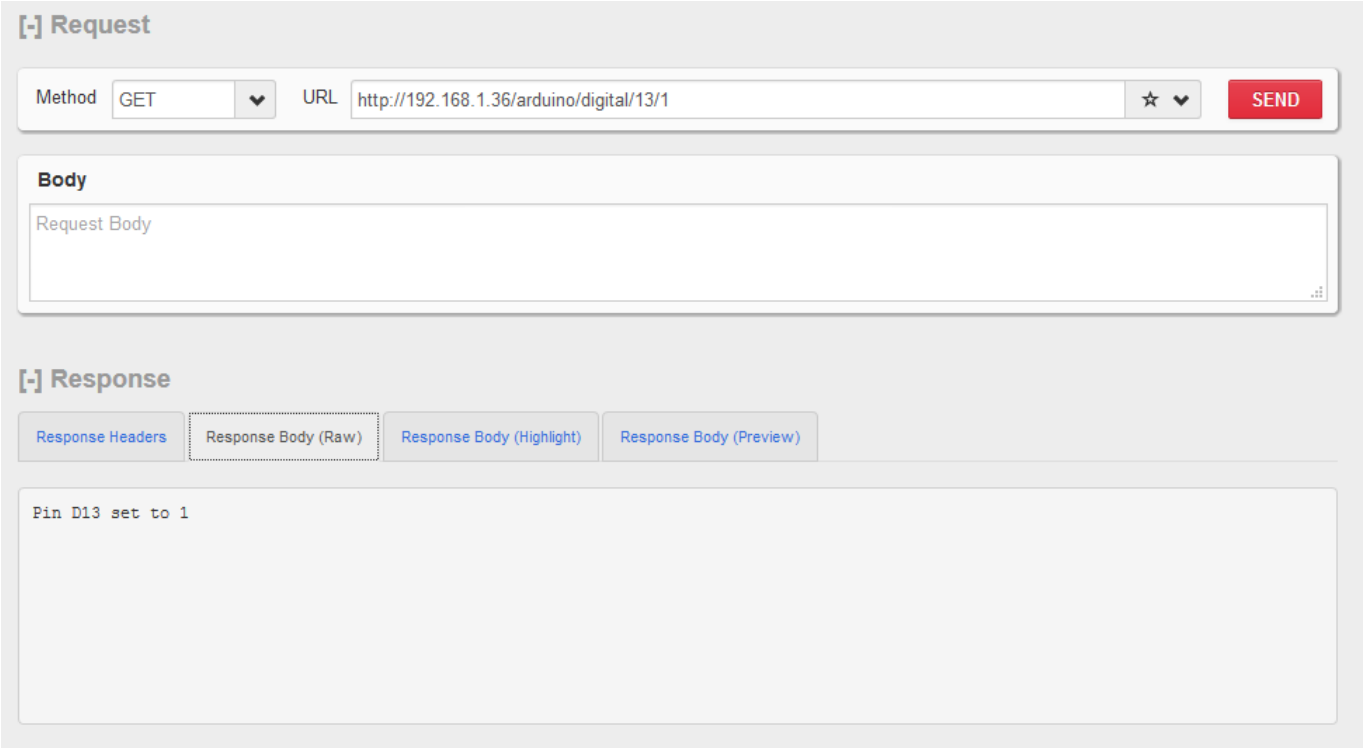
Bien lo siguiente sera subir un código de ejemplo al Arduino para poder testear la aplicación. Para más información sobre

como subir un programa des del IDE a nuestro Arduino podéis mirar este post Arduino – Primera Parte. Utilizaremos un ejemplo de la librería **Bridge.**



Una vez subido nuestro programa atacaremos la Api Rest para encender y apagar el led que vine incorporado en la placa, exactamente el 13 (situado justo detrás del puerto microusb). En esta caso utilizo el plugin para Firefox RESTclient, aunque podríamos utilizar cualquier otro. En la url especificamos la ip de nuestro Seeeduino, arduino(podemos acceder a otros

elementos de la placa), el tipo de pin (en este caso digital), el pin (13) y el valor (0 apagado, 1 encendido). La url final sería **http://192.168.1.36/arduino/digital/13/1** (encendido) o **http://192.168.1.36/arduino/digital/13/0** (apagado)



Conclusión

Como podemos ver la potencia de Seeeduino Cloud es mucha. Fácilmente podemos acceder a nuestro Arduino recibiendo y enviando valores a través de una sencilla Rest Api. En el próximo post acoplaremos un Relay Shield, esto nos permitirá controlar remotamente aparatos electrónicos (dado el calor que hace ahora, un pequeño ventilador)

Android – Butterknife

Introducción

En este post trataremos con una librería muy útil. En este caso nos ayuda deshacernos de mucho código. De esta manera nuestra aplicación nos queda mucha más limpia y legible. Lo mejor de todo es que esta librería inyecta código en tiempo de compilación, es decir, el rendimiento de nuestra aplicación no se verá afectado por su uso.

- Configuración
- Ejemplo de uso con Activity
- Ejemplo de uso con Fragment
- Eventos
- Usando Resources
- Plugin ButterKnife Zelezny

Configuración

Para la instalación, tenemos que añadir el plugin android-apt a nuestro classpath en el fichero build.gradle. Este se encuentra en la raíz de nuestro proyecto.

```
dependencies{
  classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
}
```

Dentro del archivo app/build.gradle, debemos añadir el plugin antes de añadir las dependencias de Butterknife.

```
apply plugin : 'com.neenbedankt.android-apt'

dependencies {
  compile 'com.jakewharton:butterknife:8.0.1'
  apt 'com.jakewharton:butterknife-compiler:8.0.1'
}
```

Ejemplo de uso

Eliminaremos el uso de findViewById utilizando @BindView en los views de Android (TextView, Button, EditText…):

```
class MainActivity extends Activity {
  // Automatically finds each field by the specified ID.
  @BindView(R.id.title) TextView title;
  @BindView(R.id.name) EditText name;
  @BindView(R.id.btn_send_name) Button sendName;

  @Override public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_activity);
    ButterKnife.bind(this);
    // Some code
  }
}
```

Como podemos ver se simplifica el código y a su vez se hace

más legible.

Ejemplo de uso con Fragment

Cuando utilicemos fragments tendremos que especificar en el método bind la vista con la que vamos a trabajar y en el evento onDestroyView utilizar unbind:

```
public class SimpleFragment extends Fragment {
  @BindView(R.id.txt_name) Button name;
  @BindView(R.id.btn_send_name) Button sendName;

  @Override public View onCreateView(LayoutInflater inflater,
ViewGroup container, Bundle savedInstanceState) {
     View view = inflater.inflate(R.layout.simple_fragment,
container, false);
    ButterKnife.bind(this, view);
    // Some code
    return view;
  }

  // When binding a fragment in onCreateView, set the views to
null in onDestroyView.
   // Butter Knife has an unbind method to do this
automatically.
  @Override public void onDestroyView() {
    super.onDestroyView();
    ButterKnife.unbind(this);
  }
}
```

Eventos

Los eventos serán funciones con la anotación correspondiente.

```
@OnClick(R.id.submit)
public void sayHi(Button button) {
  button.setText("Hello!");
}
```

También podemos agrupar vistas y asignarlas a un único evento

```
@OnClick({R.id.maint_btn_change_text,
R.id.main_btn_new_intent})
    void buttonClick(View v) {
        switch (v.getId()){
            case R.id.maint_btn_change_text:
                title.setText(name.getText().toString());
                break;
            case R.id.main_btn_new_intent:
                 Intent resourcesIntent =  new Intent(this,
ResourcesActivity.class);
                startActivity(resourcesIntent);
                break;
        }
    }
```

Podremos utilizar los siguientes eventos: OnClick, OnLongClick, OnEditorAction, OnFocusChange, OnItemClick, OnItemLongClick,OnItemSelected, OnPageChange, OnTextChanged, OnTouch, OnCheckedChanged.

Usando resources

Podemos hacer binding de resources fácilmente

```
@BindString(R.string.title) String title;
@BindDrawable(R.drawable.my_drawable) Drawable myDrawable;
@BindColor(R.color.red) int red;
```

Plugin ButterKnife Zelezny

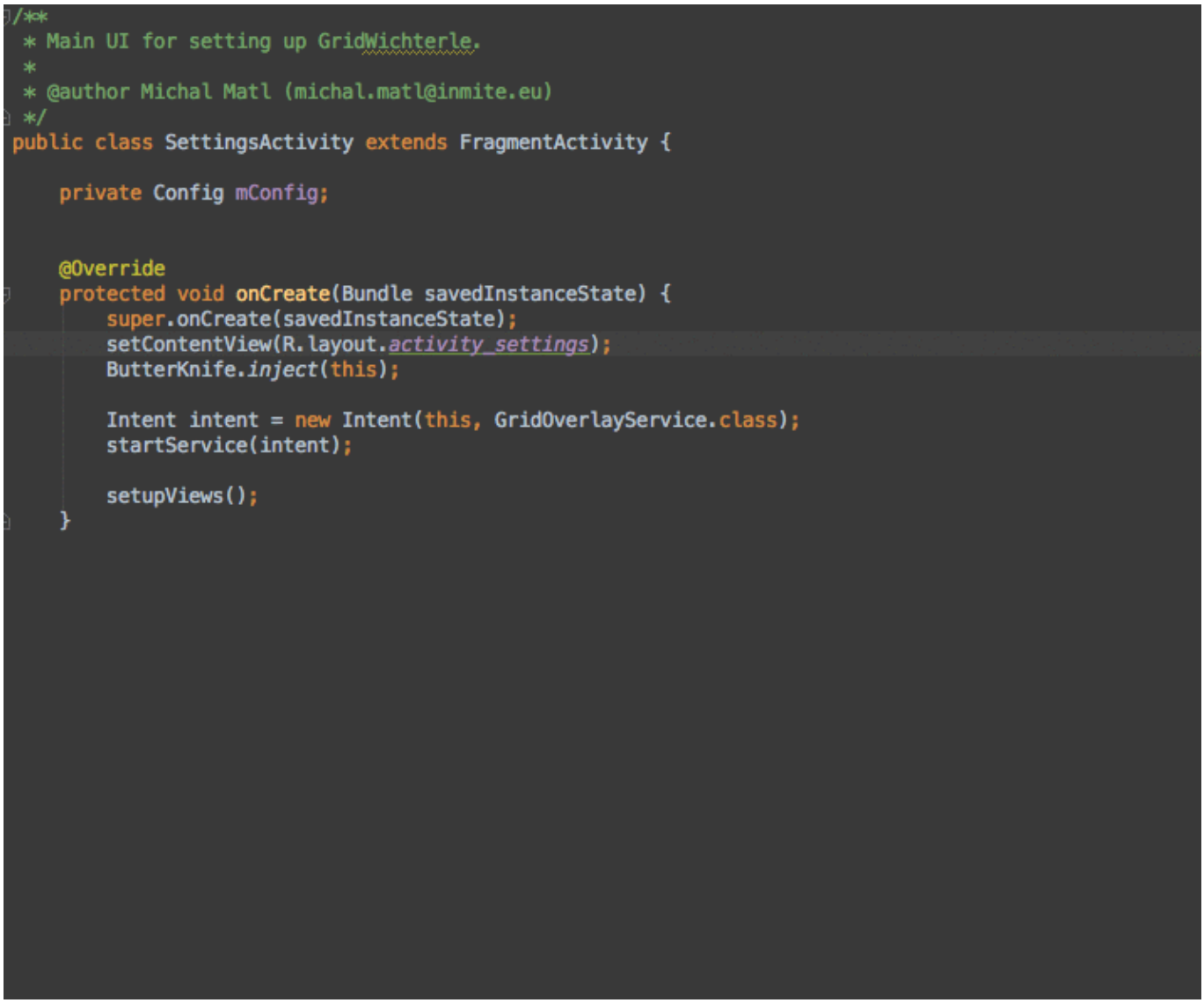
Este es un plugin para Android Studio muy útil. A partir de una vista nos genera todos los bindviews para esta.

Primeramente deberemos instalar el plugin en Android Studio:

- Des de Android Studio: iremos a **File** -> **Settings** -> **Plugins** -> **Browse repositories** y buscaremos **ButterKnife Zelezny**

- Descargandolo: decargamos el plugin ButterKnife Zelezny y lo instalamos des de **File** -> **Settings** -> **Plugins** -> **Install plugin from disk**

Para hacer uso del plugin debemos incluir Butterknife tal y como hemos mostrado al principio del post. Finalmente, os dejo un gif sacado de la página del proyecto de GitHub donde se muestra su utilización.



```
/**
 * Main UI for setting up GridWichterle.
 *
 * @author Michal Matl (michal.matl@inmite.eu)
 */
public class SettingsActivity extends FragmentActivity {

    private Config mConfig;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_settings);
        ButterKnife.inject(this);

        Intent intent = new Intent(this, GridOverlayService.class);
        startService(intent);

        setupViews();
    }
```

Como nota final, decir que se generan los nombres de las variables a partir de los ids asignados a las vistas del layout seleccionado. Es decir, **main_btn_show_toast** se convertirá en **mainBtnShowToast**. En el cuadro de dialogo podremos modificar esos nombres, uno a uno claro. Si

observamos atentamente un correcto «naming» en nuestro layout nos generará automáticamente variables con un «naming» adecuado. El problema es que nos encontramos con la anotación correspondiente al layout, **main**BtnShowToast. Solucionar esto es fácil, cogeremos como ejemplo las variables correspondientes a los Buttons del layout main:

- Seleccionamos el layout y la primera letra correspondiente a la vista que queremos modificar

```
@BindView(R.id.main_btn_resources) Button mainBtnResources;
@BindView(R.id.main_btn_disable_button) Button mainBtnDisableButton;
```

- Seguidamente con atajo de teclado **alt + j** seleccionaremos todas la variables correspondientes al layout y la inicial de la vista a modificar

```
@BindView(R.id.main_btn_resources) Button mainBtnResources;
@BindView(R.id.main_btn_disable_button) Button mainBtnDisableButton;
```

- Y finalmente, corregimos el nombre para todas las variables

```
@BindView(R.id.main_btn_resources) Button btnResources;
@BindView(R.id.main_btn_disable_button) Button btnDisableButton;
```

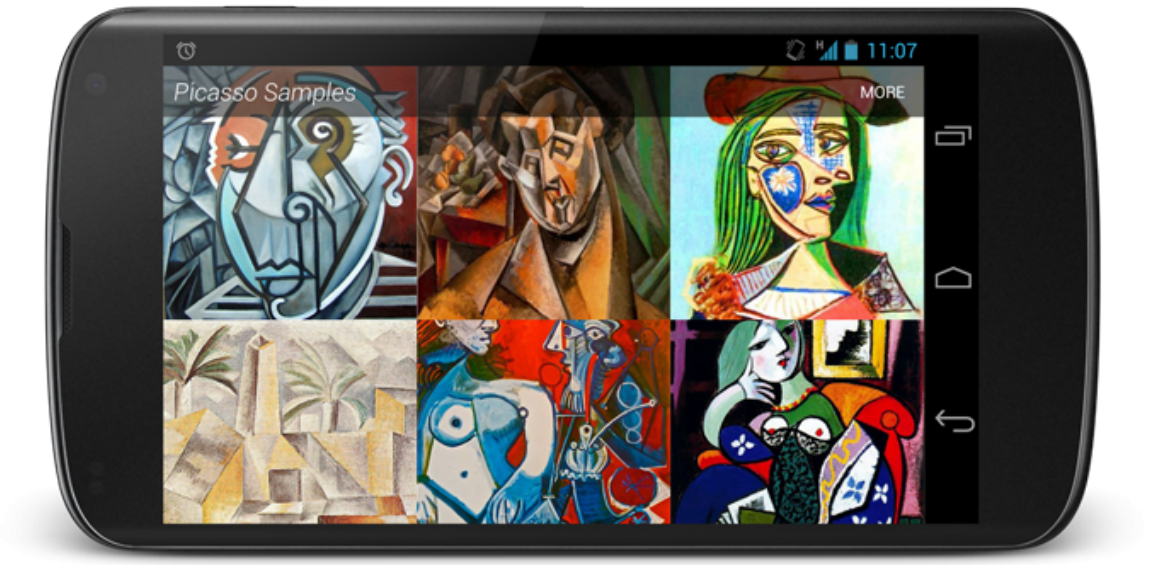
Observaciones

Butterknife nos ofrece una manera de mantener nuestro código limpio y legible. Algo que a largo plazo se hace indispensable, pues cualquier software se va a tener que mantener. Y que nuestro código sea limpio y legible es vital para esta tarea. Hemos mostrado también un plugin muy útil para utilizar la librería y al final hemos hablado un poco de naming. Esto daría para otro post y llegará. Por otro lado también hemos mostrado un atajo de teclado de Android Studio, los atajos de teclado para Android Studio dan para otro futuro post y así podríamos seguir y no parar nunca. Finalmente, dejo

el enlace a la web de Butterknife (donde podréis encontrar ejemplos más avanzados), el enlace a la cuenta de GitHub del plugin ButterKnife Zelezny y un simple ejemplo en Github (el proyecto de Picasso refactorizado usando Butterknife).

- Butterknife
- ButterKnife Zelezny
- ButterKnifeTest

Android – Picasso



Introducción

Seguimos con una de esas librerías que nos solucionan de manera espectacular la utilización de imágenes en nuestros proyectos, Picasso. Existen otras como Glide (recomendada por Google) o Fresco (Facebook). Las diferencias entre Picasso y Glide son pocas, en cuanto a Fresco se refiere, es una aproximación diferente al tratamiento de imágenes en android. En este post (nearsoft) podeis ver una breve comparación entre

la tres y en este otro post (inthecheesefactory blog) encontraras una comparación entre Glide y Picasso. Bien, vamos con Picasso:

- Configuración y utilización
- Picasso con assets o drawables
- Resize ,fit y rotation
- Scaling
- Placeholder y error
- Picasso Transformation Library
- Cache Indicators y Logging
- Observaciones

Configuración y utilización

Dentro del archivo app/build.gradle, debemos añadir la dependencia de Picasso.

```
dependencies{
  compile 'com.squareup.picasso:picasso:2.5.2'
}
```

Una vez añadida podemos empezar a utilizarla, como podéis ver es muy simple:

```
ImageView ourImageView = (ImageView) findViewById(R.id.imageView);
String remoteUrl = "http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg";

Picasso
    .with(this)
    .load(remoteUrl)
    .into(ourImageView);
```

Picasso con assets, drawables o ficheros guardados

Realmente fácil:

```
// Loading drawable
Picasso
    .with(this)
    .load(R.drawable.image)
    .into(imageView1);

// Loading asset
Picasso
    .with(this)
    .load("file:///android_asset/image.png")
    .into(imageView2);

// Loading file from storage
File file = new File(Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES), "Android.png");
Picasso
    .with(this)
    .load(file)
    .into(imageView3);
```

Resize, fit y rotation

Podemos redimensionar la imagen.

```
Picasso
    .with(this)
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg")
    .resize(100, 100)
    .into(imageView);
```

Pero en caso de que la imagen sea más pequeña tenemos la opción de evitar este reescalado. Redimensionar una imagen haciéndola más grande nos supone un uso de recursos que muchas veces nos va a dar un resultado muy pobre. En este caso podemos utilizar **scaleDown(true)**, de esta manera la imagen se redimensionará si el resultado final implica unas dimensiones inferiores a las originales.

```
Picasso
    .with(this)
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .resize(100, 100)
    .scaleDown()
    .into(imageView);
```

O bien hacer que se redimensione automáticamente al tamaño del imageView. Esto puede hacer que la carga de la imagen tarde un poco más, puesto que primero se debe esperar a poder obtener el tamaño del imageView.

```
Picasso.with(this)
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .fit()
    .into(imageView);
```

Podemos rotar la imagen

```
Picasso.with(this)
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .rotate(180f)
    .into(imageView);
```

E incluso indicar que punto queremos utilizar para pivotar la rotación

```
// rotate(float degrees, float pivotX, float pivotY)
Picasso
    .with(this)
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .rotate(45f, 200f, 100f)
    .into(imageViewComplexRotate);
```

Scaling

El reescalado de imágenes puede afectar el aspect ratio y

hacer que esta se vea deforme. Para solucionar esto podemos utilizar centerCrop() o centerInside().

CenterCrop
Se escala la imagen haciendo que coincida con los límites del ImageView y entonces se elimina la parte restante de la imagen. El ImageView contendrá la imagen pero seguramente se perderán partes de esta.

```
Picasso
    .with(this)
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .resize(100, 100) // resizes the image to these dimensions
(in pixel)
    .centerCrop()
    .into(imageViewResizeCenterCrop);
```

CenterInside
Se escala la imagen teniendo en cuenta que las dos dimensiones de la imagen son iguales o inferiores al tamaño del imageView. La imagen se mostrara completa pero puede que no ocupe todo el imageView.

```
Picasso
    .with(this)
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .resize(100, 100)
    .centerInside()
    .into(imageViewResizeCenterInside);
```

Placeholder y error

Podemos utilizar una imagen temporal hasta que nuestra imagen se haya cargado y otra en caso de que ocurra un error. De esta manera, el usuario tendrá la sensación que durante un tiempo de espera o incluso un error la aplicación funciona perfectamente.

```
Picasso.with(this)
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .placeholder(R.drawable.placeholder_image)
    .error(R.drawable.error_image)
    .into(imageView);
```

Picasso Transformation Library

Existe una librería que nos permite hacer transformaciones a nivel avanzado y de manera muy fácil picasso-transformations. Os recomiendo que paséis por su cuenta de Github puesto que aquí solo mostraremos dos ejemplos.

Primeramente en el archivo app/build.gradle, debemos añadir la dependencia de picasso-transformations.

```
dependencies{
    compile 'jp.wasabeef:picasso-transformations:2.1.0'
}
```

Una vez añadida podemos empezar a tulizarla, por ejemplo para aplicar un crop circular:

```
    Picasso
    .with(this)

.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .transform(new CropCircleTransformation())
    .into(imageView);
```

También podemos encadenar transformaciones, por ejemplo haciendo un crop como en el ejemplo anterior y aplicando un filtro de color:

```
int color = Color.parseColor("#3300ff80");
Picasso
    .with(this)
    .load("http://www.guitarthai.com/picpost/gtpicpost/Q367224
```

```
.jpg")
    .transform(new ColorFilterTransformation(color))
    .transform(new CropCircleTransformation())
    .into(imageView);
```

Cache Indicators y Logging

Cache Indicators
Picasso utiliza dos tipos de chache, memoria y almacenamiento. En algunos casos, para comprobar y hacer tests de rendimiento de nuestra aplicación, nos interesará saber de donde se ha obtenido la imagen. Para hacerlo debemos añadir la opción **.setIndicatorsEnabled(true)**

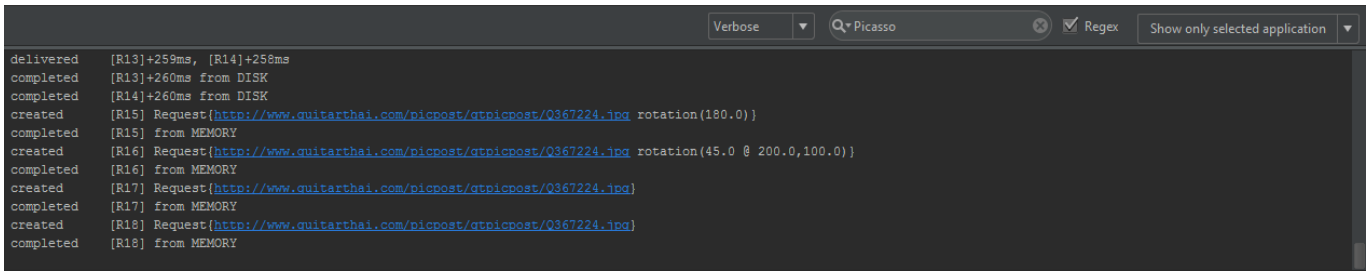
```
Picasso
    .with(this)
    .setIndicatorsEnabled(true);
Picasso
    .with(this)
    .setIndicatorsEnabled(true);
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .into(imageView);
```

Una vez cargadas la imágenes estas tendrán un indicador de color en la parte superior izquierda. El esquema de colores corresponde al origen del cual la imagen es:

- Verde (memoria, mejor rendimiento)
- Azul (memoria interna del dispositivo, rendimiento medio)
- Rojo (red, el peor rendimiento)

Logging
Aveces necesitamos más información que una simple indicación del origen de la imagen. Utilizando la opción **.setLoggingEnabled(true);** obtendremos en el log una información más acurda del proceso creado por Picasso.

```
Picasso
    .with(this)
    .setLoggingEnabled(true);
Picasso
    .with(this)
    .setLoggingEnabled(true);
.load("http://www.guitarthai.com/picpost/gtpicpost/Q367224.jpg
")
    .into(imageView);
```



Observaciones

Bien, hoy hemos tratado una librería la cual conviene tener en la caja de herramientas. Nos ofrece muchas opciones y nos permite realizar operaciones de una manera muy simple. Como contrapartida podemos decir que el rendimiento siempre se verá afectado. Finalmente, dejo el enlace a la web de Picasso y un simple ejemplo en Github.

- Picasso
- PicassoTest

Android – Parceler

Parceler

Android Parcelable code generator for Google Android

Introducción

Como se comento en el último post de Java para android en adelante trataremos librerías y ejemplos de novedades, para cursos de android podéis encontrar enlaces en el post anterior. En dicho post en la parte de serialización comentábamos que en android existían parcelables y el porqué debíamos utilizarlos. Bien pues hoy presentamos una librería que facilita mucho su uso Parceler

- Configuración
- Utilizando Parceler en nuestra clase
- Observaciones

Configuración

Para la instalación, tenemos que añadir el plugin android-apt a nuestro classpath en el fichero build.gradle que se encuentra en la raíz de nuestro proyecto.

```
dependencies{
  classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
}
```

Dentro del archivo app/build.gradle, debemos añadir el plugin antes de añadir las dependencias de Parceler.

```
plugin : 'com.neenbedankt.android-apt'

dependencies {
  compile 'org.parceler:parceler-api:1.1.5'
  apt 'org.parceler:parceler:1.1.5'
```

```
}
```

Utilizando Parceler a nuestra clase

Incluiremos un constructor vacío, las variables de clase deben ser public y finalmente utilizaremos la anotación **@Parcel**. Bien esto nos ahorra mucho código y al mismo tiempo este se hace mucho más legible. Pero tiene un inconveniente, asume como entorno toda la clase User. Para evitar encapsular variables que no son necesaria utilizaremos la anotación **@Transient**

```
@Parcel
public class User {
    // class vars must be public
    public String name;
    public String lastName;
    public String job;
    @Transient
    public boolean hasCar;

    // empty constructor needed by the Parceler library
    public User() {
    }

     public User(String name, String lastName, String job,
boolean hasCar) {
        this.name = name;
        this.lastName = lastName;
        this.job = job;
        this.hasCar = hasCar;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getJob() {
        return job;
    }

    public void setJob(String job) {
        this.job = job;
    }

    public boolean isHasCar() {
        return hasCar;
    }

    public void setHasCar(boolean hasCar) {
        this.hasCar = hasCar;
    }
}
```

Para crear un parcelable utilizaremos **Parcels.wrap**:

```
User user = new User("Joan", "Miquel", "Android Developer",
true);
Intent intent = new Intent(getApplicationContext(),
SecondActivity.class);
intent.putExtra("user", Parcels.wrap(user));
```

Para recuperar el objeto user utilizaremos **Parcels.unwrap**:

```
user = Parcels.unwrap(getIntent().getParcelableExtra("user"));
```

Observaciones

Como podéis comprobar el uso de esta librería es muy útil y ademas muy fácil incluir en nuestros proyectos. Pero no todo es perfecto, como toda librería que utilicemos, mejorará con el tiempo y solucionara bugs. A priori esto es bueno pero no disponemos de un sistema de upgrade automático. Debemos de estar detrás de las novedades y preocuparnos notros de actualizarla. Tenemos que tener en cuenta que utilizar esta librería tiene sus ventajas e inconvenientes. Nos facilita mucho el uso de parcelabe pero al mismo tiempo es menos eficiente que implementarlo nosotros mismos. De echo el uso de esta librería es un buen balance entre eficiencia y facilidad de uso en comparación con una serialización y una implementación de parcelabe. Cabe destacar que si nuestra aplicación requiere de la máxima eficiencia lo mejor será implementar parcelable. A continuación os dejo el enlace a la página de la librería, un simple ejemplo en GitHub y un artículo en android developers de como usar parcelables

- Parceler
- ParcelerTest
- Parcelable (Android Developer)

Curso de Java orientado a Android – Parte 4

Curso de Java orientado a Android

Introducción

En este último post del curso de Java orientado a Android trabajaremos un poco mas a fondo la programación orientada a objetos y algunas particularidades propias de Java. Este va ser el último post, de esta serie.

- Nested classes
- Beneficios de las inner classes
- Variables de clase (static)
- Funciones (static)
- Enumerated types
- Serialization
- Deserializing

Nested classes

En Java se puede definir una clase dentro de otra clase. Estas se llaman «nested class»:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Las nested class pueden ser static:

```
class OuterClass {
    ...
    static class StaticInnerClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

Una nested class es miembra de su outer class. Las nested clases no estáticas tienen acceso a otros miembros de la clase externa, incluso si se declaran como private. Sin embargo, las nested static clases no. De manera similar a las variables y métodos miembros, una clase interna pueden ser declarados privado, público, protegido, o un paquete privado.

Beneficios de utilizar inner class

Las siguientes son algunas de las razones que tientan a un programador para usar clases internas:

- **Mejorar la agrupación lógica de clases** que se utilizan únicamente en un solo lugar. Si una clase B es útil solamente a otra clase A, entonces es lógico que la clase B sea una clase interna de la clase A.
- **Aumentar la encapsulación**. Si la clase B necesita tener acceso a los miembros privados de la clase A, un programador puede ocultar la clase B dentro A y mantener todos los miembros de la clase A como privados al mismo tiempo que oculta la clase B del resto del entorno.
- **Mejorar la legibilidad del código y facilita el mantenimiento**. La creación de las clases internas dentro de una clase externa proporciona una organización más clara de código.

Variables de clase (static)

Cuando creamos varios objetos de la misma clase, cada objeto (instancia) tiene su propia copia de las variables miembro. A veces, puede ser que deseemos compartir una variable con todos los objetos de la misma clase. Para lograr esto usamos modificador static.

Las variables miembro que tienen el modificador static en su declaración se llaman campos estáticos o variables de clase staticas. Están asociadas con la clase, en lugar de con cualquier objeto. Cada instancia de la clase comparte una variable de clase, que se guarda en una memoria fija. Cualquier objeto puede cambiar el valor de una variable de clase.

Vamos a modificar la clase de coches de la parte 2 de esta serie de posts añadiendo una variable de clase static. La variable numOfSeats puede tener valores diferentes para los distintos objetos del tipo de coche. Sin embargo, podemos añadir una variable de clase llamada numberOfCars que se utilizará para realizar un seguimiento del número de objetos de coches creados.

```
  public class Car extends Vehicle {
    public int numOfSeats;
    //A class variable for the
    //number of Car objects created
    public static int numberOfCars;
  }
```

Las variables de clase son referenciadas por el propio nombre de la clase:

```
Car.numberOfCars;
```

Funciones de clase (static)

Java también soporta métodos estáticos, estos tienen el modificador static en su firma. Un uso común de los métodos estáticos es tener acceso a los campos estáticos. Por ejemplo, vamos a modificar la clase de coche mediante la adición de un método estático que devuelve la variable statica numOfCars:

```
  public static int getNumberOfCars() {
    return numberOfCars;
  }
```

Enumerated types

Un enumerated type (también llamado enumeration o enum) es un tipo de datos que consiste en un conjunto de constantes llamadas elementos. Un ejemplo común de enumeración es los días de la semana. Dado que son constantes, los nombres de los campos de un enum están en letras mayúsculas.
Para definir un tipo de enum en Java, utilizamos la palabra clave enum. Por ejemplo, el siguiente tipo de enumeración define un conjunto de enumeraciones para las versiones de Android:

```
public enum androidVersionCodes {
    CUPCAKE, DONUT,ECLAIR,FROYO,
    GINGERBREAD,HONEYCOMB,ICE CREAM SANDWICH,
    JELLY BEAN,KITKAT,LOLLIPOP,MARSHMALLOW
}
```

Los enum se deben utilizar siempre que se necesite representar un conjunto fijo de constantes.

Serialization

La serialización es el proceso de convertir un objeto en un formato que puede ser almacenado y luego convertido de nuevo más tarde a un objeto en el mismo o en otro entorno informático.

Java proporciona serialización automática, para tulizarla el

objeto debe implementar la interfaz java.io.Serializable . Java entonces maneja serialización internamente.

La siguiente es una clase Java que almazena una versión de android (nombre y codigo de versión). Es serializable, y tiene dos variables miembro: androidVersionName y androidVersionCode.

```
import java.io.Serializable;

public class AndroidVersions implements Serializable {
    private Double androidVersionCode;
    private String androidVersionName;

    public Double getAndroidVersionCode () {
        return androidVersionCode;
    }

          public void setAndroidVersionCode(Double
androidVersionCode) {
        this.androidVersionCode = androidVersionCode;
    }

    public String getAndroidVersionNmae() {
        return getAndroidVersionNmae;
    }

          public void setAndroidVersionName(String
getAndroidVersionNmae) {
        this.getAndroidVersionNmae = getAndroidVersionNmae;
    }
}
```

Ahora que tenemos un objeto serializable, podemos hacer una prueba del proceso de serialización escribiendo un objeto en un fichero. El siguiente código escribe un objeto AndroidVersions en un fichero llamado android.ser:

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
```

```
import java.util.ArrayList;

public class SerializeAndroidVersions {

    public static void main(String[] args) {

        AndroidVersions andVersions = new AndroidVersions();
        andVersions.setName("Jelly Bean");
        andVersions.setAndroidVersionCode(4.1);

        try
        {
                        FileOutputStream fileOut = new
FileOutputStream("/home/user_name/android.ser");
                         ObjectOutputStream out = new
ObjectOutputStream(fileOut);
            out.writeObject(e);
            System.out.println("Serialized...");
            out.close();
            fileOut.close();
        }
        catch (IOException i) {
            i.printStackTrace();
        }
    }
}
```

Como resultado tendremosun fichero android.ser en nuestra carpeta de usuario.

Deserializing

Ahora podemos crear un objeto a partir del fichero guardado en disco. Lo único que debemos hace es acceder al fichero y convertirlo en un objeto.
El siguiente ejemplo muestra como realizar este proceso:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
```

```
import java.util.ArrayList;

public class DeserializeAndroidVersions {
  @SuppressWarnings(
    "unchecked"
  )

  public static void main(String[] args) {
    AndroidVersions androidVersions = new AndroidVersions();
    try{
                    FileInputStream   fileIn   =   new
FileInputStream("/home/_user_name/android.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
      androidVersions = (AndroidVersions) in.readObject();
      in.close();
      fileIn.close();
    }catch (IOException i) {
      return;
    }catch ClassNotFoundException c) {
      System.out.println("AndroidVersions class not found.");
      c.printStackTrace();
      return;
    }

    if(androidVersion instanceof AndroidVersions) {
      System.out.println("-----------------------");
         System.out.println("Deserialized AndroidVersions
object...");
                     System.out.println("Name:    "    +
androidVersions.getAndroidVersionNmae());
              System.out.println("Code  version:  "  +
androidVersions.getAndroidVersionCode());
      System.out.println("-----------------------");
    }
  }
}
```

Ejecutando el código anterior obtendremos el siguiente resultado:

```
-----------------------
Deserialized AndroidVersions object...
```

Name: Jelly Bean
Code version: 4.1

Llegados a este punto ahora entendemos la lógica de la serialización. Esta bien a nivel conceptual, de aprendizaje, pese a ello este procedimiento no es el usual en Android. Podemos utilizar la serialización, pero en vez de ello utilizaremos Parcelable. En este artículo se explica el porqué debemos utilizar parcelabes.

Observaciones

Con este post damos por terminada la introducción a la programación en Java para Android. La idea era realizar un curso de Android una vez finalizada esta serie, pero no va a ser así. Des de que empezó la serie hasta ahora viendo como se esta desarrollando el entorno Android he optado por no hacerlo. Me centraré ejemplos concretos y en librerías. Exsisten muschas y ayudan muchissímo en el dia a dia de la programación en Android. Tareas que a priori pueden parecer muy complicadas o introducir excesivo código nos las facilitan. No por eso voy a hacer el salto sin más, partiendo de lo aprendido anteriormente, a continuación dejo tres muy buenos recursos para aprender Android:

- Android Official Training (Inglés)
- Vogella tutorials (Inglés)
- Sgoliver curso Android (Castellano)

Curso de Java orientado a Android – Parte 3

Curso de Java orientado a Android

Introducción

Después de cubrir los conceptos básicos de los principios de la programación orientada a objetos en el post anterior, ahora atenderemos a otros conceptos que nos encontraremos a la hora de programar en Java. Exactamente como tratar la gestión de errores y colecciones de información.

- Exceptions
- Java Collections
    - Interfaces
    - Implementations
        - ArrayList
        - HashSet
        - HashMap

Exceptions

La gestión de errores es una parte esencial para asegurar un código robusto. Java utiliza excepciones para manejar errores. Cuando se produce un error, el entorno de ejecución de Java maneja un objeto de error que se crea por el método en el que se produce el error. Este objeto se llama excepción, y contiene información básica sobre el error (como el tipo de error, la ubicación, la pila de los métodos que conducen al error … etc.).
La lista de los métodos que conducen al error se le llama call stack (pila de llamadas). Al manejar el error, el sistema busca a través de esa pila un controlador de errores en el código; es decir, un controlador de excepciones. Todos los objetos de excepción son los hijos de la clase padre Exception.

Las Excepciones en Java se pueden clasificar en dos tipos:

Categoría	**Descripción**
Excepción controlada	Estos son errores dentro de código de la aplicación. Un programador que tiene la intención de crear un código bien escrito y robusto debe poder recuperar el programa de estos errores. Por ejemplo, leyendo un archivo en disco, un programador debe tener en cuenta que el fichero quizás no exista. En este caso, el programador debe esperar una excepción del tipo java.io.FileNotFoundException. Debera, enconces, capturar para notificar al usuario de lo que ha ocurrido de una forma correcta y controlada sin detener la ejecución del programa.

Excepción no controlada	Estos vienen en dos tipos: los propios errores y tiempo de ejecuciónexcepciones. Se agrupan en una categoría porque ambosno es posible anticipar o recuperarse de un programador.Los errores son externos a las aplicaciones. Por ejemplo, supongamosque una aplicación abre correctamente un archivo para la entrada, pero es no puede leer el archivo debido a un hardware o del sistemaFuncionamiento defectuoso. La lectura infructuosa arrojarájava.io.IOError, y tiene sentido para que el programaimprimir un seguimiento de pila y salir. Los errores son esas excepciones de tipoClase de fallo y sus subclases.Excepciones de tiempo de ejecución generalmente indican errores de programación,como errores lógicos. Son o una clase de tipo RuntimeException ysus subclases.

La gestión de errores (excepciones) en Java se realiza a través de los bloques try-catch-finally . Mientras que la bloque finally es opcional, los bloques try y catch son obligatorios para cumplir con el tratamiento de errores.
Veamos el siguiente código:

```
 public class ExceptionTest {
        public static void main(String[] args) {
                        System.out.println("Hello World
adictosalainformatica!");
               String nullString = null;
               System.out.println("Entered try statement");
                                String partialString =
nullString.substring(1);
                // Execution will break before reaching this
line
                 System.out.println("Partial string is: " +
partialString);
        }
 }
```

El resultado de ejecutar el código será un error del tipo NullPointerException, en concreto en la línea 6, donde estamos tratando de leer de un objeto de cadena que es nulo (no inicializado) Para manejar adecuadamente este error, debemos modificar el código anterior de la siguiente manera:

```
public class ExceptionTest {
        public static void main(String[] args) {
                        System.out.println("Hello World
adictosalainformatica!");
                String nullString = null;
                try {
                          System.out.println("Entered try
statement");
                                    String partialString =
nullString.substring(1);
                          // Execution will continue in the
exception block
                        System.out.println("Partial string is:
"+partialString);
                } catch (Exception e) {
                          System.out.println("Error occured:
"+e.getMessage());
                        e.printStackTrace();
                }
        }
}
```

En vez de romper la ejecución de código y detener el programa, este código se encargará de manejar la excepción NullPointerException mediante la impresión de los detalles del error y continuando la ejecución más allá de la bloque catch. El bloque finally se puede utilizar después del bloque de excepción. Este bloque de código se ejecutará siempre, tanto si se lanza una excepción como no.

```
try {
        System.out.println("Entered try statement");
        String partialString = nullString.substring(1);
        // Execution will break before reaching this line
           System.out.println("Partial string is: " +
partialString);
} catch (Exception e) {
        System.out.println("Error occured: "+e.getMessage());
        e.printStackTrace();
} finally {
         System.out.println("This line of code will always
```

```
run!");
}
```

En muchos casos utilizamos el bloque finally cuando hay algunos recursos que necesitan ser liberados. Es posible que después de una excepción no lo esto no ocurra porque el código que debería hacerlo no es ejecutado. Por ejemplo, un programa que lee un fichero, este debe cerrar el archivo después de terminar la lectura y/o escritura. Si una excepción es lanzada, la línea de código que cierra el archivo puede ser omitida. El bloque finally sería el mejor lugar donde cerrar el fichero.

Java Collections

Java proporciona un conjunto de clases e interfaces para ayudar a los desarrolladores manejar colecciones de objetos. Esta colección de clases són similiares a un array, excepto por su tamaño (puede crecer de forma dinámica durante tiempo de ejecución). En esta sección se ofrecerá una visión general de algunas de las clases de Java collection más populares.

Interfaces

Las Java Collections se encuentran principalmente en el paquete java.util. Proporciona dos interfaces principales: Collection y Map. Estas dos forman el núcleo del Java Collection framework. Otras interfaces heredan de estas dos. Por ejemplo, las interfaces List y Set heredan de la interfaz Collection. Todas estas interfaces son genéricas; es decir, el tipo de objeto contenido en la colección debe ser especificado por el programador. Hay una diferencia fundamental entre las subclases de la interfaz Collection y de la interfaz Map:

- Collection contiene un grupo de objetos que pueden ser manipulados y traspasados. Los elementos pueden ser

duplicados o únicos, dependiendo del tipo de sub-clase. Por ejemplo, Set sólo contiene objetos únicos.

- La interfaz Map, sin embargo, mapea los valores a los ids (keys) y no puede contener keys duplicadas y a cada una sólo se le puede asignar un valor

Implementations

Las implementaciones son los objetos de datos que se utilizan para almacenar colecciones, que implementan las interfaces de la sección anterior. A continuación se explican las siguientes implementaciones:

ArrayList
Un ArrayList es una implementación de array redimensionable de la interfaz List. Implementa todas las operaciones opcionales de lista y permite todo tipo de elementos, incluyendo null. También proporciona métodos para manipular el tamaño del array que se utiliza internamente para almacenar la lista.

```
import java.util.*;

class TestArrayList {
        public static void main(String args[]) {
                // Creating an array list
                ArrayList androids = new ArrayList();
                // Adding elements
                androids.add("Cupcake");
                androids.add("Donut");
                androids.add("Eclair");
                androids.add("Froyo");
                androids.add("Gingerbread");
                androids.add("Honeycomb");
                androids.add("Ice Cream Sandwich");
                androids.add("Jelly Bean");
                  System.out.println("Size of ArrayList: " +
androids.size());
                // Display the contents of the array list
                  System.out.println("The ArrayList has the
following elements: "
```

```
                + androids);
                // Remove elements from the array list
                    System.out.println("Deleting second
element...");
                androids.remove(3);
                 System.out.println("Size after deletions: " +
androids.size());
                 System.out.println("Contents after deletions:
" + androids);
        }
}
```

El resultado será:

```
Size of ArrayList: 8
The ArrayList has the following elements: [Cupcake, Donut,
Eclair, Froyo,
Gingerbread, Honeycomb, Ice Cream Sandwich, Jelly Bean]
Deleting second element...
Size after deletions: 7
Contents after deletions: [Cupcake, Donut, Eclair,
Gingerbread, Honeycomb,
Ice Cream Sandwich, Jelly Bean]
```

HashSet

Esta clase implementa la interfaz Set y permite el elementos null. Esta colección no permite duplicados. Crea una colección que utiliza una tabla hash para el almacenamiento. Una tabla almacena información de hash mediante el uso de un mecanismo llamado hashing donde se utiliza el valor almacenado para determinar una clave única, que se utiliza como el índice en el que los datos se almacenan. La ventaja del hashing es que permite tiempos de ejecución rápidos para operaciones básicas, como add() y remove().

```
class TestHashSet {
        public static void main(String args[]) {
                // Creating a HashSet
                HashSet androids = new HashSet();
                // Adding elements
```

```
                androids.add("Cupcake");
                androids.add("Cupcake");
                androids.add("Eclair");
                androids.add("Eclair");
                androids.add("Gingerbread");
                androids.add("Honeycomb");
                androids.add("Ice Cream Sandwich");
                androids.add("Jelly Bean");
                  System.out.println("The contents of the
HashSet: " + androids);
        }
}
```

El resultado será:

The contents of the HashSet: [Eclair, Cupcake, Honeycomb, Ice Cream Sandwich, Jelly Bean, Gingerbread]

Como podemos comprobar solo hay un elemento “Cupcake” y uno “Éclair” aunque en el código los hayamos añadido dos veces.

HashMap

Esta es una hashtable basada en la implementación de la interface Map. Los elementos introducidos no tendrán un orden específico y permite elementos null.

El siguiente programa muestra un ejemplo de HashMap. Se asignan nombres para contabilizar saldos.

```
import java.util.*;
class TestHashMap {
        public static void main(String args[]) {
                // Creating a HashMap
                   HashMap<String,Double> androids = new
HashMap<String,Double>();
                // Adding elements
```

```
                androids.put("Cupcake", new Double(1.5) );
                androids.put("Donut",new Double(1.6));
                androids.put("Eclair", new Double(2.1));
                androids.put("Froyo", new Double(2.2));
                androids.put("Gingerbread", new Double(2.3));
                androids.put("Honeycomb", new Double(3.1));
                   androids.put("Ice Cream Sandwich", new
Double(4.0));
                androids.put("Jelly Bean", new Double(4.1));
                // Get a set of the entries
                   Set<Map.Entry<String, Double>> set =
androids.entrySet();
                // Get an iterator
                  Iterator<Map.Entry<String, Double>> i =
set.iterator();
                // Display elements
                while (i.hasNext()) {
                          Map.Entry<String, Double> me =
(Map.Entry<String,Double>)
                       i.next();
                       System.out.print(me.getKey() + ": ");
                       System.out.println(me.getValue());
                }
                System.out.println();
                // Increase version number of Eclair
                Double version = androids.get("Eclair");
                 androids.put("Eclair", new Double(version +
0.1));
                  System.out.println("New version number of
Eclair: "
                + androids.get("Eclair"));
        }
}
```

El resultado será:

```
Eclair: 2.1
Cupcake: 1.5
Honeycomb: 3.1
Froyo: 2.2
Donut: 1.6
Ice Cream Sandwich: 4.0
```

Jelly Bean: 4.1
Gingerbread: 2.3
New version number of Eclair: 2.2

Observaciones
En este tercer post hemos mostrado el manejo de excepciones y la utilización de Collections. Se ha mostrado una parte específica y concreta, sería recomendable profundizar un poco más en estos conceptos dado que, aquí solo se muestran conceptos básicos.

- Oracle Exception tutorial
- Oracle Collections tutorial

Curso de Java orientado a Android – Parte 2

Introducción

Java es un lenguaje de programación orientado a objetos (OOP).

En este post cubriremos las características y principios básicos de la programación orientada a objetos proporcionando algunos ejemplos de código. Este post puede completarse con los siguientes: Pincipios de programación orientada a objetos y Principios del diseño de clases

- Objects
- Classes
- Getters y Setters
- Inheritance
- Keywords this y super
- Interface
- Access Modifiers
- Contructors
- Method overriding and overloading
- Polymorphism

Objects

Un objeto es un conjunto de software de estado y el comportamiento relacionado en memoria. Normalmente se utilizan para representar objetos del mundo real. Los objetos son esenciales para la comprensión OOP. Los objetos del mundo real comparten dos características: estado y el comportamiento. Por ejemplo, un coche tiene un estado (modelo actual , fabricante, color) y el comportamiento (conducción, cambio de marchas …)

El desarrollo de aplicaciones con código orientado a objetos aporta muchos beneficios, incluyendo código de fácil reutilización, ocultación de información, facilidad de depuración…

Classes

Una clase es un prototipo a partir de la cual se crean los objetos. En ella se definen los modelos – estado y el comportamiento de un objeto del mundo real. Las clases proporcionan una forma limpia para modelar el estado y comportamiento de los objetos del mundo real. Podemos distinguir dos propiedades, o secciones, principales a definir en una clase:

- Un conjunto de variables de clase (también llamadas campos)
- Un conjunto de métodos de clase (o funciones).

Para representar el estado de un objeto en clases tenemos las variables de clase. Los comportamientos de los objetos son representados usando métodos. La siguiente es una sencilla clase Java llamada de vehículo.

```
class Vehicle {
        int speed = 0;
        int gear = 1;
        void changeGear(int newGear) {
                gear = newGear;
        }
        void speedUp(int increment) {
                speed = speed + increment;
        }
        void printStates() {
                  System.out.println(" speed:" + speed + "
gear:" + gear);
        }
}
```

El estado del objeto Vehículo se representa con la velocidad (speed) y las marchas (gear). El comportamiento del objeto se puede cambiar utilizando los dos métodos ChangeGear() y speedUp(). Por último podemos saber cual es el estado actual con el método printStates()

Getters y Setters

Un conjunto de métodos se crean por lo general en una clase para leer/escribir los valores de las variables miembro . Estos son llamados **getters**(se utiliza para obtener los valores) y **setters**(se utiliza para cambiar la valores de variables miembro).

Los Getters y Setters son cruciales en las clases de Java, ya que se utilizan para gestionar el estado de un objeto. En la clase vehículo que hemos visto anteriormente, podemos añadir dos métodos (un getter y un setter) para cada variable miembro. El siguiente es el código completa la clase anteriror con los getters y setters correspondientes a la variables miembro speed y gear:

```
class Vehicle {
        int speed = 0;
        int gear = 1;
        // Start of getters and setters
        public int getSpeed() {
                return speed;
        }
        public void setSpeed(int s) {
                speed = s;
        }
        public int getGear() {
                return gear;
        }
        public void setGear(int g) {
                gear = g;
        }
        // End of getters and setters
        void changeGear(int newGear) {
                gear = newGear;
        }
        void speedUp(int increment) {
                speed = speed + increment;
        }
        void printStates() {
```

```
                System.out.println(" speed:" + speed + "
gear:" + gear);
        }
}
```

Inheritance

La herencia proporciona un mecanismo poderoso y natural para la organización y estructuración del software. Se establece una relación padre-hijo entre dos objetos diferentes.

La programación orientada a objetos permite que las clases hereden estado y comportamiento de uso común de otras clases. En siguiente ejemplo, vehículo (Vehicle) se convierte en la clase padre (superclass) de camiones (Truck) y coches (Car). En el lenguaje de programación Java, permite que cada clase tenga una superclase directa, y cada superclase puede ser heredada por un número ilomitado de subclases.

```
public class Car extends Vehicle {
        int numOfSeats;
        //Set of statements defining
        //a car’s state and behavior
}

public class Truck extends Vehicle {
        public int loadWeight;
        //Set of statements defining
        //a truck’s state and behavior
}
```

Ahora la clase Truck y Car comparten el estado y comportamiento de la clase Vehicle.

Keywords this y super

Dos palabras clave de Java que se puede encontrar al escribir que el código de clase con la herencia: **this** y **super**. La

palabra clave **this** se utiliza como una referencia a la clase actual, mientras que **super** es una referencia a la clase padre que esta clase ha heredado. En otras palabras, **super** se utiliza para acceders a las variables y métodos miembro de la clase padre.
La palabra reservada **super** es especialmente útil cuando se desea reemplazar el método de la superclase en la clase hijo, pero se desea invocar el método de la superclase. Por ejemplo, en la clase de Car, se puede sobrescribir el método printStates() y a su vez llamar al método printStates de la clase Vehicle ():

```
public class Car extends Vehicle {
        int numOfSeats;
        void printStates() {
                super .printStates();
                  System.out.println(" Number of Seats:" +
numOfSeat);
        }
}
```

Llamando al método printStates() de la clase Car se invocará primero printStates() de la clase Vehicle y posteriormente se mostrara el resultado de println.

Interface

Una interfaz es un contrato entre una clase y el mundo exterior. Cuando una clase implementa una interfaz, debe proporcionar el comportamiento especificado por esa interfaz. Tomando el ejemplo de Vehicle crearemos una interfaz.

```
public interface IVehicle {
        void changeGear(int newValue);
        void speedUp(int increment);
}
```

Ahora la clase Vehicle implementa la interfaz IVehicle utilizando la sintaxis siguiente:

```
class Vehicle implements IVehicle {
        int speed = 0;
        int gear = 1;
        public void changeGear(int newValue) {
                gear = newValue;
        }
        public void speedUp(int increment) {
                speed = speed + increment;
        }
        void printStates() {
                  System.out.println(" speed:" + speed + "
gear:" + gear);
        }
}
```

Se debe tener en cuenta que la clase de Vehicle debe proporcionar y la implementación de métodos los métodos changeGear() y speedUp().

Access Modifiers

Los modificadores de acceso determinan si otras clases pueden utilizar una variable o invocar un método en concreto. Hay cuatro tipos de control de acceso:

- A nivel de clase – public o default (sin modificador explícito).
- A nivel de miembro de clase (variable o método) – public, private, protected o default (sin modificador explícito).

Una clase puede ser declarada como pública con el modificador public, en cuyo caso esa clase es visible para todas las clases en todas partes. Si una clase no tiene modificador (por defecto), es visible sólo dentro de su propio package (un package es una agrupación de clases afines).
A nivel de miembro, además los modificadores public o default (package-private), existen dos modificadores de acceso

adicionales: private y protected. El modificador private especifica que el miembro sólo puede ser accedido des de su propia clase. El modificador protected especifica se puede acceder al miembro dentro de su propio package y, además, por cualquier otra subclase de de esta.

Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**AllOther**
public	**Y**	**Y**	**Y**	**Y**
protected	**Y**	**Y**	**Y**	**N**
Default	**Y**	**Y**	**N**	**N**
private	**Y**	**N**	**N**	**N**

Contructors

Los constructores se invocan para crear objetos. Son similares a las funciones, pero diferenciadas por la siguiendo:

- Los constructores tienen el mismo nombre que la clase
- No tienen ningún tipo de retorno.

Llamar a un constructor para crear un nuevo objeto sería inicializar miembros de un objeto. suponer El vehículo tiene el siguiente constructor:

```
public Vehicle(int speed, int gear){
        this.speed = speed;
        this.gear = gear;
}
```

Para crear un nuevo objeto de la clase Vehicle invocando el constructor deberemos utilizar la palabra reservada new:

```
Vehicle vehicle = new Vehicle(4, 2);
```

Esto creará un nuevo objeto de la clase Vehicle con sus dos

variables de clase speed y gear inicializadas a 4 y 2.

Method overriding and overloading

Dentro de la misma clase, se pueden crear dos métodos con el mismo nombre pero se diferencia en el número de argumentos y sus tipos. Esto se llama sobrecarga de métodos.
La sobrecarga de métodos ocurre cuando una clase hereda un método de una super clase, pero ofrece su propia aplicación de ese método. En el siguiente código, la clase Car sobrecarga el método SpeedUp() definido en la clase de Vehicle.

```
public class Car extends Vehicle {
        int numOfSeats;
        public void speedUp(int increment) {
                speed = speed + increment + 2;
        }
}
```

Supongamos que creamos un objeto de tipo Car y llamamos al método llamó a la speedUp(). Entonces, el método del Vehicle es ignorado y se ejecuta el de la clase Car:

```
Car car = new Car();
car.speedUp(2);
```

Polymorphism

En el contexto de la programación orientada a objetos, el polimorfismo significa que las diferentes subclases de la misma clase padre puede tener comportamientos diferentes, pero comparten algunas de las funcionalidades de la clase padre.
Para demostrar el polimorfismo, añadiremos el método showInfo () a la clase de Vehicle. Este método imprime toda la información en un objeto del tipo de Vehicle:

```
public void showInfo() {
         System.out.println("The vehicle has a speed of: " +
```

```
this.speed
                + " and at gear " + this.gear);
}
```

Sin embargo, si la subclase Truck utiliza este método, la variable miembro loadWeight no será imprimida, ya que no es un miembro de la clase padre Vehicle. Para resolver esto, podemos sobreescrivir el método showInfo() de la siguiente manera:

```
public void showInfo() {
        super.showInfo();
         System.out.println("The truck has is carrying a load
of: "
                + this.loadWeight);
}
```

Podemos observar que el método showInfo() de Truck, llamará showInfo() de la clase padre y agregar a ella su propio comportamiento – el cual imprime el valor de loadWeight.
Podemos hacer lo mismo con la clase Car.

```
public void showInfo() {
        super.showInfo();
        System.out.println("The car has "
                + this.numOfSeats + " seats.");
}
```

Ahora, podemos hacer un test de polimorfismo. Crearemos 3 objetos, cada uno de un tipo de Vehicle diferente

```
class TestPolymorphism {
        public static void main(String[] args) {
                Vehicle vehicle1, vehicle2, vehicle3;
                vehicle1 = new Vehicle(50,2);
                vehicle2 = new Car(50,2,4);
                vehicle3 = new Truck(40,2,500);
                System.out.println("Vehicle 1 info:");
                vehicle1.showInfo();
                System.out.println("\nVehicle 2 info:");
                vehicle2.showInfo();
                System.out.println("\nVehicle 3 info:");
                vehicle3.showInfo();
```

```
        }
}
```

El resultado de ejecutar esta clase será la creación de tres tipos de objeto diferentes:

```
Vehicle 1 info:
The vehicle has a speed of: 50 and at gear 2
Vehicle 2 info:
The vehicle has a speed of: 50 and at gear 2
The car has 4 seats.
Vehicle 3 info:
The vehicle has a speed of: 40 and at gear 2
The truck has is carrying a load of: 500
```

En el ejemplo anterior, la JVM ha llamado el método de cada objeto en lugar de llamar el objeto de Vehicle.

Observaciones

En este segundo post hemos introducido varios conceptos de programación orientada a objetos enfocada al entorno Java. Esta parte es muy importante, se debe tener en cuenta que un correcto diseño nos va asegurar en el futuro un fácil mantenimiento del código. Bien porqué tengamos que añadir funcionalidades o porqué debemos solucionar bugs. Esta parte a nivel teórico se puede ampliar con los siguientes posts:

- Pincipios de programación orientada a objetos
- Principios del diseño de clases