

LifeCycles and Intents (Parte I)

Ciclo de Vida de una actividad

Competencias:

- Reconocer etapas de vida de una actividad
- Aplicar etapas de vida de una actividad

Introducción

Los ciclos de vida de una actividad representan los distintos estados que tiene esta misma actividad durante el uso de una aplicación android, en otras palabras, con los métodos del ciclo de vida puedes declarar como será controlado el acceso o cierre de una actividad por parte del usuario.

Es muy importante aprender a usar los métodos del ciclo de vida de una actividad para desarrollar aplicaciones de calidad y para brindar una mejor experiencia al usuario final.

Conceptos del ciclo de vida de una actividad

Cuando creamos una clase en nuestra aplicación al extender dicha clase de nuestra super clase Activity, por ejemplo, tenemos acceso a seis métodos que cumplen con controlar los estados de la actividad durante su uso, los cuales son el onCreate, onStart, onResume, onPause, onStop y el onDestroy. En este contexto la declaración de estos métodos indicando su protección y tipo de retorno, es de la siguiente forma:

```
public class Activity extends Activity {  
    protected void onCreate(Bundle savedInstanceState);  
  
    protected void onStart();  
  
    protected void onResume();  
  
    protected void onPause();  
  
    protected void onStop();  
  
    protected void onDestroy();  
}
```

Métodos de estado de una actividad

Cada uno de estos métodos cumple una función importante en el ciclo de vida de nuestra actividad, así que veamos el detalle de cada uno de ellos a continuación:

onCreate(): El método onCreate suele ser implementado por los programadores, aunque no es obligatorio, se le pudiera interpretar como tal, ya que desde este método usualmente se inicializan e instancian tanto los archivos de diseño (layouts), como las vistas asociadas al layout, para poder ser precargadas en nuestra pantalla de actividad. Este método onCreate() inicia cuando el sistema operativo crea la actividad en la aplicación.

Así se ve un método onCreate():

```
@Override
public void onCreate(Bundle savedInstanceState) {
    // call the super class onCreate to complete the creation of activity like
    // the view hierarchy
    super.onCreate(savedInstanceState);

    // recovering the instance state
    if (savedInstanceState != null) {
        gameState = savedInstanceState.getString(GAME_STATE_KEY);
    }

    // set the user interface layout for this activity
    // the layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);

    // initialize member TextView so we can manipulate it later
    textView = (TextView) findViewById(R.id.text_view);
}
```

onStart(): al momento en que el método onCreate() finaliza sus instrucciones, comienza la etapa de hacer visible la actividad al usuario, y esto ocurre desde el método onStart(), inicializando el código precargado desde el onCreate() y las instrucciones que hayan sido escritas en este onStart(), para que nuestro hilo principal UI (User Interface) reciba y mantenga las instrucciones dadas.

```
@Override
protected void onStart() {
    super.onStart();
    Toast.makeText(getApplicationContext(), "Ha iniciado nuestra actividad en la
    UI", Toast.LENGTH_SHORT).show();
}
```

onResume(): este método es invocado por defecto después del método onStart(), como lo indica la imagen 1 que describe el ciclo de vida de una actividad. Por otra parte, este estado también ocurre después de haber existido un evento onStop, que detuviese la actividad, por ejemplo, el haber minimizado la aplicación para abrir otra, representa esta situación, en la cual, al volver a maximizar nuestra aplicación, la actividad residente ejecuta el método onResume, recuperando todas las declaraciones detenidas.

```
@Override
public void onResume() {
    super.onResume();
    Toast.makeText(getApplicationContext(),"Hemos vuelto a la actividad",Toast.LENGTH_SHORT).show();
}
```

onPause(): La actividad llama este método como primer indicio de que el usuario está abandonando nuestra actividad, a este llamado le seguirá automáticamente el del método onStop. Este método de pausa del ciclo de vida lo puedes usar indicar que por ejemplo, un servicio que esté trabajando no debe continuar su operación.

```
@Override
public void onPause() {
    super.onPause();
    Toast.makeText(getApplicationContext(),"Se ha pausado nuestra actividad",Toast.LENGTH_SHORT).show();
}
```

onStop(): este método es invocado cuando la actividad ya no es visible para el usuario, quedando en un estado de detención pero sin ser descartada por completo, dado que el acto seguido a una actividad detenida es un recuperarla o cerrarla completamente, ambas acciones son ejecutadas con el método onResume() y onDestroy() respectivamente.

```
@Override
public void onStop() {
    super.onStop();
    Toast.makeText(getApplicationContext(),"Se ha detenido nuestra actividad",Toast.LENGTH_SHORT).show();
}
```

onDestroy(): este método realiza operaciones de limpieza de por ejemplo objetos instanciados, vistas desplegadas en ui, para liberar en la aplicación los recursos de memoria que usaba la actividad. Este método también puede ser invocado por el sistema operativo en caso de errores o limitaciones de memoria para liberar espacio.

```
@Override
public void onDestroy() {
    super.onDestroy();
    Toast.makeText(getApplicationContext(),"Se ha destruido nuestra actividad",Toast.LENGTH_SHORT).show();
}
```

En la siguiente imagen podemos visualizar todos los estados del ciclo de vida de una actividad con amplitud de vista general.

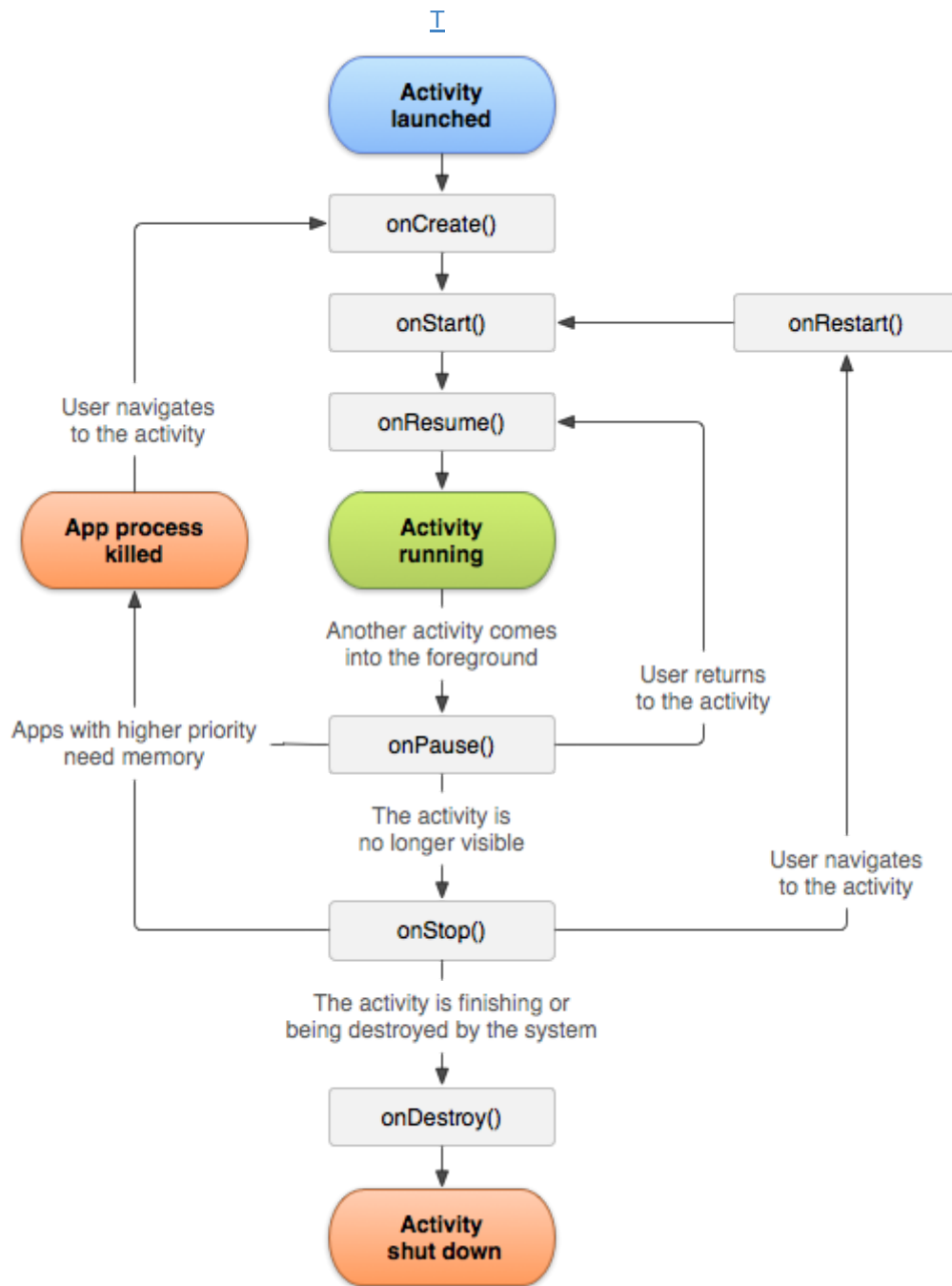


Imagen 1: Estados del ciclo de vida

Ejercicio 1:

Crear un proyecto llamado “Prueba Dinámica” con una actividad vacía, declarando los métodos del ciclo de vida en la actividad principal.

1. Iniciamos android studio.

Seleccionamos un nuevo proyecto con una actividad vacía como lo muestra la imagen 2.

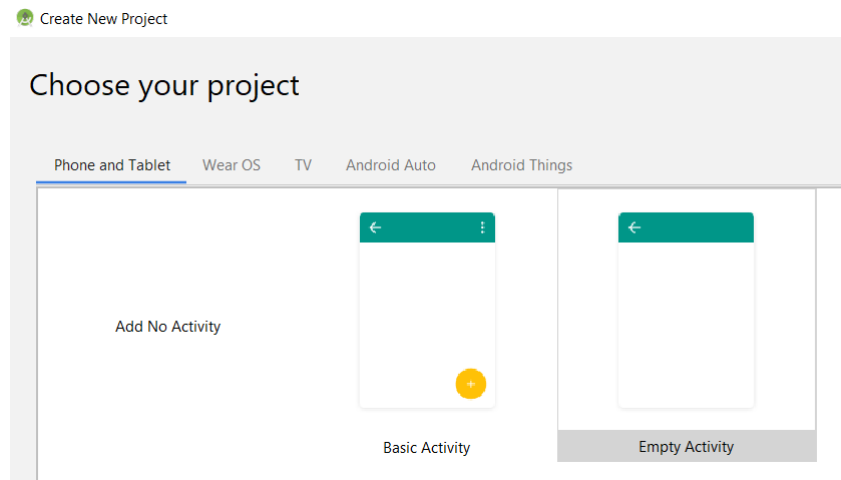


Imagen 2: Crear nuevo proyecto

3. A continuación le damos el nombre al proyecto de “Prueba Dinámica” y seleccionamos los parámetros que muestra la imagen 3.

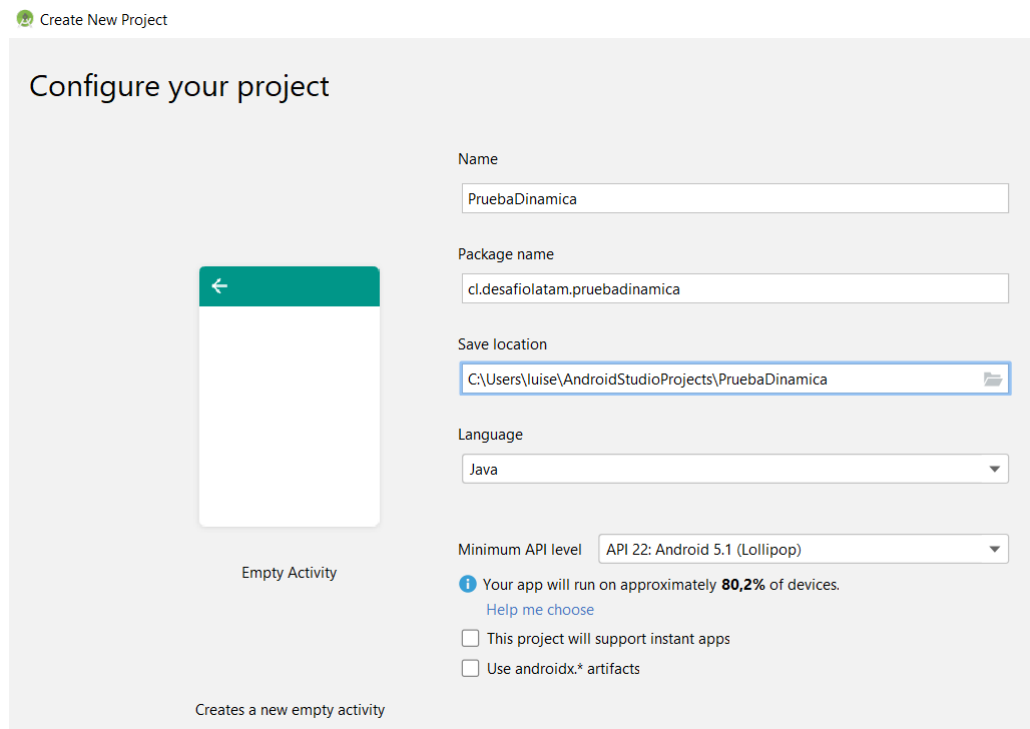


Imagen 3: Configurar nuevo proyecto

4. Teniendo nuestro proyecto ya generado, vamos a nuestra MainActivity.java y declaramos todos los métodos del ciclo de vida, el resultado debe ser el siguiente: **package** cl.desafiolatam.pruebadinamica;

```
package cl.desafiolatam.pruebadinamica;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

    @Override
    protected void onStart() {
        super.onStart();
    }

    @Override
    protected void onPause() {
        super.onPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
    }

    @Override
    protected void onStop() {
        super.onStop();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }
}
```

Toast - Mensajes de aviso nativos

La clase Toast, nos provee la función de mostrar un aviso de información simple sobre una acción en una pequeña ventana emergente. Solo ocupa la cantidad de espacio necesario para el mensaje, y la actividad en curso permanece visible y admite la interacción. Los avisos desaparecen automáticamente después de un tiempo. Ver imagen 4 de referencia.

Para utilizar el widget, crea una instancia de un objeto [Toast](#) con uno de los métodos [makeText\(\)](#). Este método toma tres parámetros: el valor [Context](#) de la app, el mensaje de texto y la duración del aviso. Devuelve un objeto de aviso correctamente inicializado. Puedes mostrar el aviso con [show\(\)](#), como lo muestra el siguiente ejemplo:

```
Toast.makeText(getApplicationContext(),"onResume: Hemos vuelto a la actividad",  
Toast.LENGTH_SHORT).show();
```

Del código anterior, destacamos:

- Que el método `getApplicationContext()` es parte de la clase `Context` de android, donde su objetivo es devolver el contexto global de la aplicación en curso. La clase `Context` tiene la finalidad de facilitar el acceso a recursos y archivos que son parte de una o varias aplicaciones a distintos niveles.
- El `Toast.LENGTH_SHORT` representa una unidad de tiempo de vida para la visualización del mensaje en nuestra pantalla.

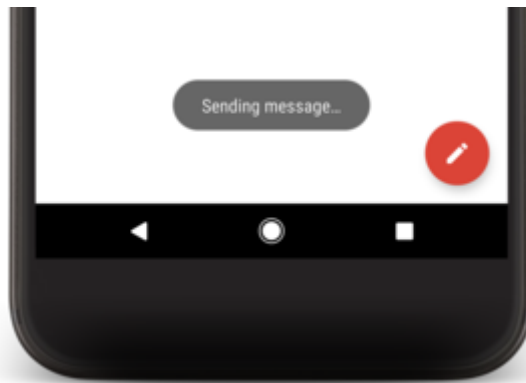


Imagen 4: Mensaje de aviso nativo

Log - Monitor

Android studio incluye una herramienta de monitoreo y seguimiento de eventos denominado Logcat, el cual te permite declarar mediante la clase `Log`, mensajes de sistema para hacerle seguimiento a las acciones y eventos de tu aplicación. Se muestran mensajes en tiempo real y también se conserva un historial para que puedas ver mensajes más antiguos.

Es una buena práctica declarar una variable constante en nuestra actividad para asignar el denominado "TAG", que funciona como un tipo de marca al mensaje de monitoreo.

```
private static final String TAG = "LifeCycleLog";
```


Ejercicio 2:

Declarando mensajes en la MainActivity

1. Agregar mensajes Toast en los métodos onResume y onStop del ciclo de vida declarados en la MainActivity.java. Para esto es necesario importar la librería android.widget.Toast.

```
import android.widget.Toast;
```

```
Toast.makeText(getApplicationContext(),"onResume: Hemos vuelto a la actividad",  
Toast.LENGTH_SHORT).show();
```

2. Agregar logs de información a cada uno de nuestros métodos del ciclo de vida declarados en la MainActivity.java. Para ello debemos importar la librería android.util.Log.

```
import android.util.Log;
```

```
Log.i(TAG, "onCreate: Creando la actividad");
```

3. Nuestro código de la clase MainActivity debe verse de esta forma:

```
package cl.desafiolatam.pruebadinamica;  
  
import android.support.v7.app.AppCompatActivity;  
import android.os.Bundle;  
import android.util.Log;  
import android.widget.Toast;  
  
public class MainActivity extends AppCompatActivity {  
  
    private static final String TAG = "LifeCycleLog";  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        Log.i(TAG, "onCreate: Creando la actividad");  
    }  
  
    @Override  
    protected void onStart() {  
        super.onStart();  
        Log.i(TAG, "onStart: Ha iniciado la actividad");  
    }  
  
    @Override  
    protected void onResume() {  
        super.onResume();  
        Log.i(TAG, "onResume: Hemos vuelto a la actividad");  
    }  
}
```

```

        Toast.makeText(getApplicationContext(),"onResume: Hemos vuelto a la actividad",
        Toast.LENGTH_SHORT).show();
    }

    @Override
    protected void onPause() {
        super.onPause();
        Log.i(TAG, "onPause: Pausada la actividad");
    }

    @Override
    protected void onStop() {
        super.onStop();
        Log.i(TAG, "onStop: Se detuvo la actividad");
        Toast.makeText(getApplicationContext(),"onStop: Se ha detenido la actividad",
        Toast.LENGTH_SHORT).show();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.i(TAG, "onDestroy: Se destruyó la actividad");
    }
}

```

4. A continuación iniciaremos nuestra aplicación desde el menú superior seleccionamos Run -> Run App , sobre el emulador de teléfono que ya tengamos configurado. Al iniciar la aplicación comenzaremos a visualizar los logs que hemos colocado en los métodos del ciclo de vida de la actividad, como lo muestra la imagen 5.

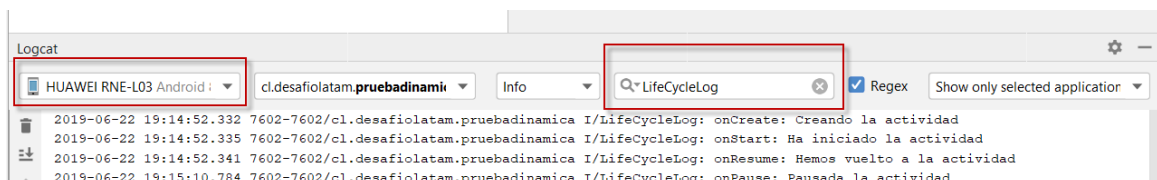


Imagen 5: Visualización de Logs

5. Para finalizar, identifiquemos los mensajes Toast que se muestran en nuestro teléfono al abrir o cerrar, minimizar o maximizar nuestra aplicación.

Integración de un API de datos externa

Competencias:

- Consumir datos de proveedores de contenido externos a través de un Api
- Utilizar datos externos en nuestras aplicaciones
- Implementar retrofit como interfaz de consumo para apis
- Desarrollar conocimientos de integración de interfaces de datos en el ciclo de vida de una actividad

Introducción

Hemos decidido incluir este capítulo “Integración de un Api de datos externa”, por ser un contenido muy valioso para un programador de aplicaciones. Hoy en día, casi el total de las aplicaciones android interactúan con algún api de datos y se conectan a internet. Han quedado en el pasado prácticamente aquellas aplicaciones standalone sin intercambio de datos.

Aprenderemos a crear una interfaz de consumo de datos externos para ser usados en nuestra aplicación, usando la librería más utilizada y recomendada por android para tal fin, retrofit.

El consumo de datos en una aplicación desde el punto de vista laboral y profesional, es la tarea más recurrente y solicitada para la construcción de aplicaciones móviles, por lo tanto, es de gran importancia tener habilidades como desarrollador para integrar datos a través de servicios web, content providers u otros.

API de datos

Un api hace referencia a un Application Programming Interface, que no es más que un grupo de métodos, funciones o servicios que ofrecen distintas aplicaciones gratuitas o no, ya sean web, móviles o de otra fuente, para ser utilizados por otra aplicación o software, es decir, un api nos provee de datos para ser utilizados en nuestros programas como capa de abstracción.

Las api son las responsables de que podamos compartir tanta información hoy en día sin la necesidad de tener acceso a bases de datos o programas ajenos.

Algunas apis de uso frecuente

- **Google Maps Api:** te permite manipular mapas mostrando ubicaciones geográficas, markers, distancias, recorridos, ciudades, países a nivel mundial.
- **Facebook Api:** muy utilizada frecuentemente como inicio de sesión seguro y rápido para usuarios.
- **YouTube Api:** acceso a canales, videos y todo lo que ya conocemos de youtube.
- **LinkedIn Api:** publicaciones, trabajos, presentaciones, vacantes, empresas.
- **Twitter Api:** micro noticias y mensajes en tiempo real.

Retrofit

Es una librería Http client para específicamente realizar peticiones http tipo rest para android y java, permitiendo el envío y recepción de respuestas de datos entre nuestras aplicaciones y distintas plataformas de software a través de internet. Retrofit simplifica la codificación para lograr esta comunicación, gestionando y transformando nuestros datos para ser traspasados a nuestras clases pojo (plain old java object).

Para realizar una petición con Retrofit se necesitará lo siguiente:

- **Una clase Retrofit:** donde se cree la instancia a la librería de retrofit para utilizar sus métodos base, definiendo aquí nuestra url base que nuestra aplicación usará para todas las peticiones http.
- **Una interfaz de operaciones:** en la cual definamos todas nuestras peticiones con sus respectivos endpoints y datos de entrada o salida.
- **Clases de modelo de datos o pojo:** para mapear los datos que llegan desde el servidor traspasando estos a objetos e instancias automáticamente para ser usados en nuestra aplicación.

Ventajas de usar retrofit

La principal ventaja de usar retrofit es el ganar tiempo de desarrollo, ya que retrofit te brinda métodos y clases que manejan muy bien las invocaciones http a servidores externos de una manera segura y eficiente. Retrofit te brinda acceso a un api de http clients.

Al no tener que escribir una librería http client por nuestra cuenta, nos olvidamos de muchos problemas como asegurar las conexiones, el caché, el manejo de requests fallidos, hilos, manejo de errores, http seguro y el parse de una respuesta.

Custom Headers

Retrofit también facilita la declaración de parámetros de cabecera cuando invocamos un método de un api externa, de manera tal, que solo tenemos que agregar una etiqueta @Headers en nuestra interfaz que declara los métodos, por ejemplo:

```
@Headers("Content-Type:application/json; charset=UTF-8")
```

GSON Converter

Habitualmente, las peticiones y respuestas son intercambiadas en formato JSON, razón por la cual se debe utilizar un convertidor junto con retrofit para lidiar con la serialización de datos en este formato. Retrofit puede deserializar por defecto respuestas de tipo OkHttp, pero admite el uso de convertidores como gson.

Gson es una biblioteca de java que nos permite convertir objetos java a json y viceversa de manera automática, sin mayores configuraciones.

Ejercicio 3:

Agregar dependencias retrofit y gson a nuestro proyecto "Prueba Dinámica".

1. Abrir el archivo build.gradle a nivel de nuestra carpeta app e incluir las dependencias de retrofit y gson de la siguiente forma:

```
implementation 'com.squareup.retrofit2:retrofit:2.6.0'  
implementation 'com.squareup.retrofit2:converter-gson:2.6.0'
```

1. Sincronizar nuestro archivo gradle.
2. Ingreseemos la siguiente url en el navegador: <https://opentdb.com/api.php?amount=10&category=15&difficulty=easy> , para visualizar la respuesta del api que a continuación vamos a mapear en nuestra aplicación con retrofit. Los datos se han de ver como nos muestra la siguiente imagen:

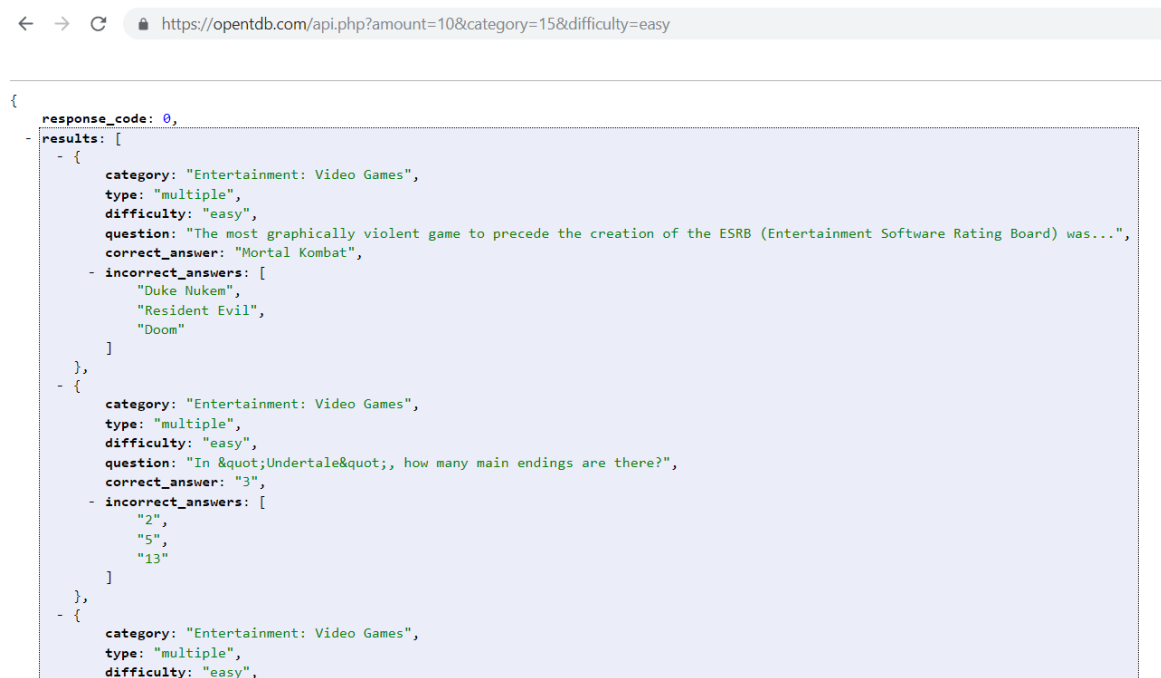


Imagen 6: Datos visualizados

Ejercicio 4:

Crear clase Retrofit basada en la api de opentdb.com, la Interfaz de operaciones y las clases para mapear json a pojo java.

1. Abrimos nuestro archivo manifest.xml y agregamos los permisos relacionados al acceso a internet, de la siguiente manera:

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

2. Nos ubicamos en nuestro package `cl.desafiolatam.pruebadinamica` y creamos un nuevo package llamado "api".
3. En el package api creamos una nueva clase java denominada "RetrofitClient" creando instancia de la librería retrofit, indicando nuestra url base de una api de datos externa denominada `opentdb.com`; y también indicamos en esta clase que el convertidor de datos a usar es gson. Nos queda de la siguiente manera:

```
package cl.desafiolatam.pruebadinamica.api;

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

public class RetrofitClient {

    private static Retrofit retrofit;
    private static final String BASE_URL = "https://opentdb.com/";

    public static Retrofit getRetrofitInstance() {
        if (retrofit == null) {
            retrofit = new retrofit2.Retrofit.Builder()
                .baseUrl(BASE_URL)
                //GSON converter//
                .addConverterFactory(GsonConverterFactory.create())
                .build();
        }
        return retrofit;
    }
}
```

4. Creamos un objeto pojo java para el mapeo de los datos a recibir desde el api opentdb en formato json. Para esto crearemos un nuevo package `cl.desafiolatam.pruebadinamica.beans`, en donde crearemos una clase llamada "PreguntasLista", la cual recibirá del objeto json el tag array "results" y el `response_code`.

```
package cl.desafiolatam.pruebadinamica.beans;

import com.google.gson.annotations.SerializedName;
import java.util.ArrayList;

public class PreguntasLista {

    @SerializedName("response_code")
    private int response_code;
    @SerializedName("results")
    private ArrayList<Pregunta> results;

    public int getResponse_code() {
        return response_code;
    }

    public void setResponse_code(int response_code) {
        this.response_code = response_code;
    }

    public ArrayList<Pregunta> getResults() {
        if(results == null){
            results = new ArrayList<>();
        }
        return results;
    }

    public void setResults(ArrayList<Pregunta> results) {
        this.results = results;
    }
}
```

5. Creamos un objeto pojo para el mapeo de los datos a recibir desde el api opentdb, ubicados en el mismo package bean, creamos una clase llamada "Pregunta", la cual recibirá todos los datos contenidos dentro del array results y mapeado en la clase "PreguntasLista".

```
package cl.desafiolatam.pruebadinamica.beans;

import com.google.gson.annotations.SerializedName;
import java.util.ArrayList;

public class Pregunta {

    @SerializedName("category")
    private String category;
    @SerializedName("type")
    private String type;
    @SerializedName("difficulty")
    private String difficulty;
    @SerializedName("questions")
    private String question;
    @SerializedName("correct_answer")
    private String correct_answer;
    @SerializedName("incorrect_answers")
    private ArrayList<String> incorrect_answers;

    public String getCategory() {
        return category;
    }
    public void setCategory(String category) {
        this.category = category;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getDifficulty() {
        return difficulty;
    }

    public void setDifficulty(String difficulty) {
        this.difficulty = difficulty;
    }

    public String getQuestion() {
        return question;
    }

    public void setQuestion(String question) {
        this.question = question;
    }

    public String getCorrect_answer() {
        return correct_answer;
    }
```



```

    }

    public void setCorrect_answer(String correct_answer) {
        this.correct_answer = correct_answer;
    }

    public ArrayList<String> getIncorrect_answers() {
        return incorrect_answers;
    }

    public void setIncorrect_answers(ArrayList<String> incorrect_answers) {
        this.incorrect_answers = incorrect_answers;
    }
}

```

6. Creamos nuestra interfaz de operaciones para los métodos de petición o request con sus respectivos endpoints.

```

package cl.desafiolatam.pruebadinamica.api;

import cl.desafiolatam.pruebadinamica.beans.PreguntasLista;
import retrofit2.Call;
import retrofit2.http.GET;

public interface Api {

    @GET("api.php?amount=10&category=15&difficulty=easy")
    Call<PreguntasLista> getAllQuestions();
}

```

7. En la clase MainActivity.java editaremos nuestro método onCreate para instanciar a la clase RetrofitClient, la interfaz de operaciones Api, haciendo uso de nuestras clase pojo para invocar el método getAllQuestions(), de la siguiente manera:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.i(TAG, "onCreate: Creando la actividad");

    Api service = RetrofitClient.getRetrofitInstance().create(Api.class);
    Call<PreguntasLista> call = service.getAllQuestions();

    //Async
    call.enqueue(new Callback<PreguntasLista>() {
        @Override
        public void onResponse(Call<PreguntasLista> call, Response<PreguntasLista> response) {
            if(response != null) {
                Log.d(TAG, response.toString());

                for(int x = 0; x < response.body().getResults().size(); x++) {
                    Log.d(TAG, "Pregunta: "+response.body().getResults().get(x).getQuestion());
                }
            }
        }
        @Override
        public void onFailure(Call<PreguntasLista> call, Throwable t) {
            Toast.makeText(getApplicationContext(), "Error: no pudimos recuperar los datos de
            opentdb", Toast.LENGTH_SHORT).show();
        }
    });
}
```

8. Iniciamos nuestra aplicación y verificamos su funcionamiento en el logcat.

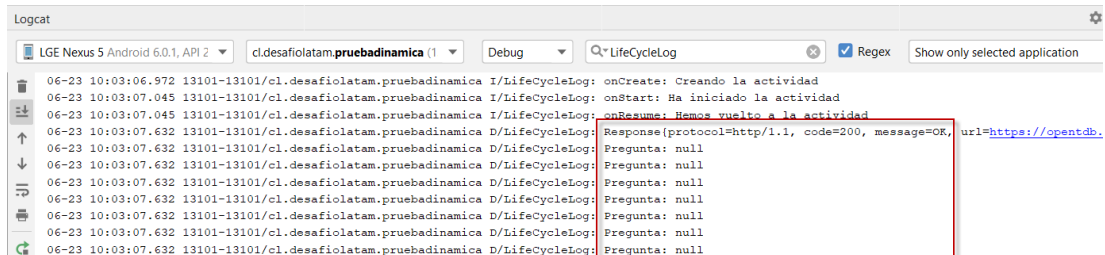


Imagen 7: Funcionamiento en el logcat

9. ¿No les funciona verdad? En el logcat solo ven resultados de Pregunta: null , pues fue es a propósito para que se enfrenten a este tipo de errores que son de los más comunes y que nos hacen sufrir por lo difícil de identificarlos en ocasiones. Para que nos funcione debemos editar nuestra clase pojo Pregunta.java modificando la siguiente línea:

```
@SerializedName("questions")
```

quedando de esta manera:

```
@SerializedName("question")
```

El error consta en que el sistema para mapear los datos correctamente debe identificar los nombres de los tags del api en su respuesta, como opendb no retorna un tag "questions" con datos sino más bien retorna un tag "question", es entonces donde debemos asegurarnos de escribir bien estas instrucciones.

10. Entonces podremos visualizar las preguntas que hemos consumido desde nuestra api a través de nuestro logcat como lo muestra la siguiente imagen:

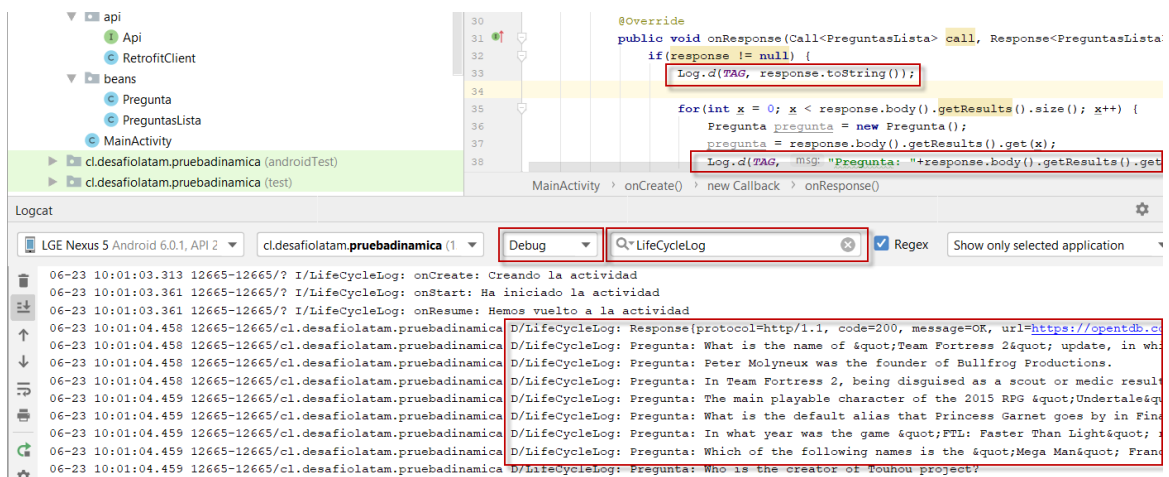


Imagen 8: Preguntas consumidas

Ciclo de Vida de un Fragmento

Competencias:

- Reconocer etapas de vida de un fragmento
- Aplicar etapas de vida de un fragmento
- Asociar un fragmento a una actividad

Introducción

Los fragmentos son componentes esenciales en el desarrollo de una aplicación android hoy por hoy. Hace algunos años, cuando las aplicaciones móviles comenzaron su auge digital, se construía una actividad por pantalla. A medida que crecían los requerimientos en las aplicaciones y la necesidad de intercambiar más y más datos, este patrón de crear múltiples actividades fue derivando en un mal desempeño del sistema y un consumo excesivo de memoria en el dispositivo móvil; es por esto que nacieron los fragmentos, componentes reutilizables que viven sobre una actividad, con el fin de mostrar múltiples pantallas sin la necesidad de crear más actividades.

Es importante para el correcto desempeño de una aplicación, la rapidez de sus acciones y el consumo limitado de recursos de memoria, aprender a trabajar con los fragmentos.

Conceptos - Ciclo de Vida de un Fragmento

Un Fragmento representa un comportamiento o una parte de la pantalla en una Actividad. Puedes combinar múltiples fragmentos en una sola actividad para crear una interfaz con secciones independientes y volver a usar un fragmento en múltiples actividades. Puedes pensar en un fragmento como una sección modular de una actividad que tiene su ciclo de vida propio, recibe sus propios eventos de entrada y que puedes agregar o quitar mientras la actividad se esté ejecutando.

Métodos de estado de un fragmento

Cuando creamos una clase en nuestra aplicación al extender dicha clase de nuestra super clase Fragment, por ejemplo, tenemos acceso a once métodos que cumplen con controlar los estados del fragmento durante su uso, los cuales son el `onAttach`, `onCreate`, `onCreateView`, `onActivityCreated`, `onStart`, `onPause`, `onResume`, `onStop`, `onDestroyView`, `onDestroy`, `onDetach`, . En este contexto la declaración de estos métodos indicando su protección y tipo de retorno, es de la siguiente forma:

```
package cl.desafiolatam.pruebadinamica.fragments;

import android.content.Context;
import android.os.Bundle;
import android.support.annotation.NonNull;
import android.support.annotation.Nullable;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class PreguntaFragment extends Fragment {

    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
    }

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {
        return super.onCreateView(inflater, container, savedInstanceState);
    }

    @Override
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
    }

    @Override
    public void onStart() {
        super.onStart();
    }

    @Override
    public void onPause() {
        super.onPause();
    }

    @Override
    public void onResume() {
        super.onResume();
    }
}
```

```

    }

    @Override
    public void onStop() {
        super.onStop();
    }

    @Override
    public void onDestroyView() {
        super.onDestroyView();
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
    }

    @Override
    public void onDetach() {
        super.onDetach();
    }
}

```

La clase Fragment tiene un código que se asemeja bastante a una Actividad. Contiene métodos de ciclo de vida similares a los de una actividad, entre los cuales están el método onCreate(), onStart(), onPause(), onResume(), onStop(), onDestroy(). A continuación detallaremos los nuevos métodos que son parte del ciclo de vida de un fragmento:

- **onAttach():** método inicial que asocia el fragmento a su contexto declarado dentro de la aplicación, esto ocurre antes de que sea iniciado el método onCreate().
- **onCreateView():** cuando el fragmento debe diseñar su interfaz de usuario por primera vez es invocado este método. Para diseñar una vista para tu fragmento, debes devolver esta vista a la actividad desde este método, el cual será el responsable de que podamos visualizar nuestros elementos. Uno de los parámetros más destacados es el inflater, el cual podemos interpretar como el responsable de inyectar un diseño de pantalla a la actividad en curso para que esta reemplace las vistas visibles por las nuevas vistas declaradas en el diseño del fragmento.

```

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.example_fragment, container, false);
}

```

- `onActivityCreated()`: método utilizado cuando los fragmentos de una actividad han sido creados y las vistas instanciadas. Se puede usar declaraciones finales justo antes de partir con la acción en la pantalla.
- `onDestroyView()`: es muy parecido al `onDestroy` pero este solo desecha un layout en curso, es decir el fragmento se mantiene en memoria, pudiendo ser recuperado con nuevas vistas.
- `onDetach()`: lo contrario al `onAttach()`, una vez eliminado nuestro fragmento, se elimina su referencia en el contexto con este método.

La siguiente imagen nos ofrece una vista general de este ciclo de vida del fragmento:

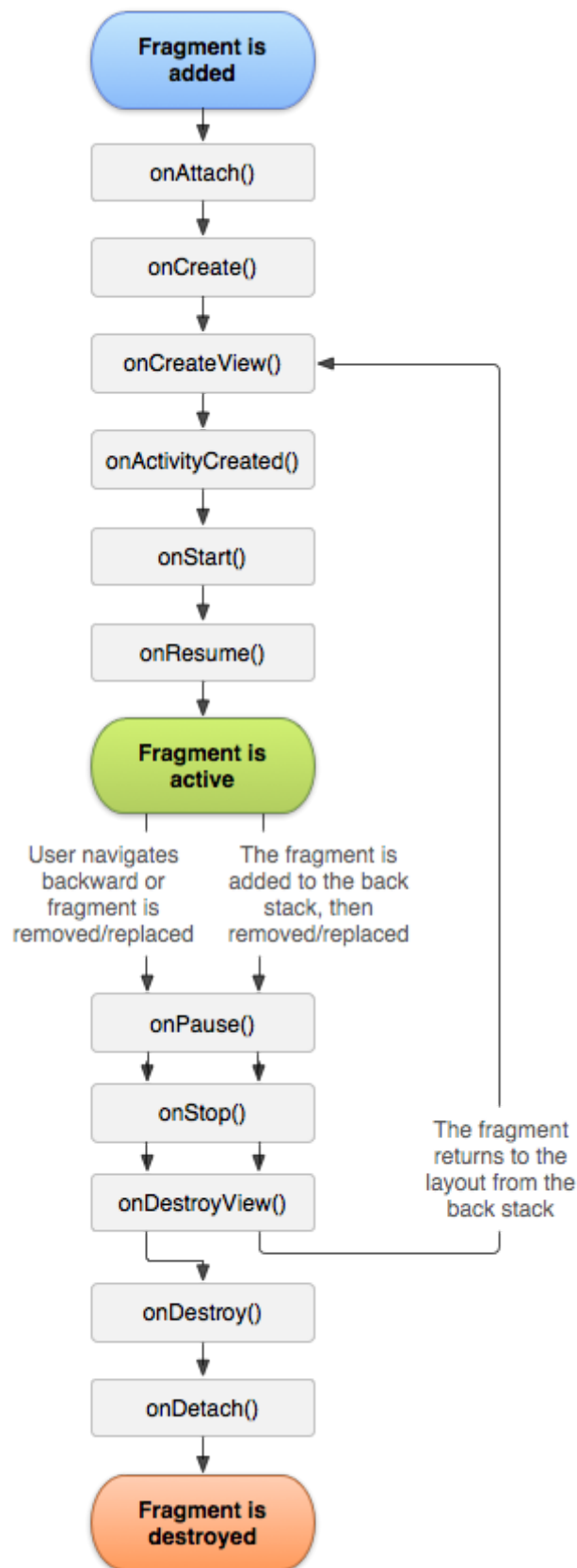


Imagen 9: Vista general del ciclo de vida del fragmento

Ejercicio 5:

Crear `FragmentoPregunta`. En adelante a nuestro proyecto general, agregaremos un nuevo fragmento a nuestro proyecto, aplicando los conocimientos adquiridos en la primera unidad referidos a vistas, grupos de vistas y actividades.

- Crearemos un nuevo layout de nombre “`fragment_pregunta.xml`” con las siguientes vistas:
- Una `ImageView` para mostrar el logo de desafío latam. Descargar e incluir la imagen en la carpeta “`drawable`” del proyecto.

<https://desafiolatam.com/assets/home/logo-academia-bla-790873cdf66b0e681dfbe640ace8a602f5330bec301c409744c358330e823ae3.png>

- `TextView` para mostrar que número de pregunta contiene el fragmento. En este ejercicio agregaremos el texto “Pregunta 1” a esta vista.
- Un `TextView` para la descripción de la pregunta.
- Un `TextView` tipo título llamado “Categoría”.
- Un `TextView` que señale la descripción de la categoría.
- Un `RadioGroup` que contenga cuatro (4) `radiobuttons`.
- Un Botón cuyo texto sería “Ver Respuesta”.

La imagen 10 representa la pantalla requerida con las vistas a construir. Los espacios en blanco enmarcados en color rojo, constituyen aquellas vistas cuyos datos llenaremos a partir de los otorgados por la api consultada en el capítulo anterior.

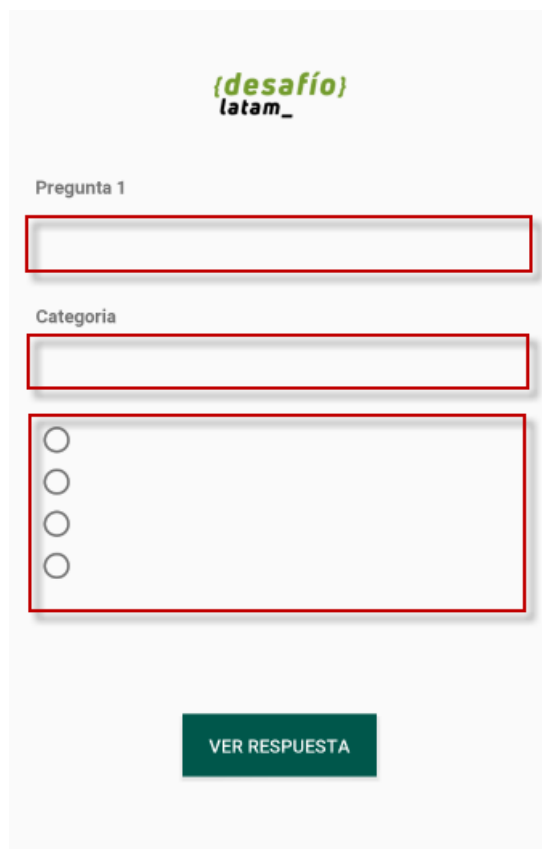


Imagen 10: Pantalla requerida con las vistas a construir

Es importante tener en cuenta el uso del viewgroup ConstraintLayout y hacer el esfuerzo de posicionar las vistas como lo muestra la imagen 10, además de considerar los márgenes y paddings que se puedan identificar en ella.

El código de la vista puede ser el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:padding="20dp"
    android:layout_marginBottom="30dp"
    android:layout_marginTop="30dp"
    >

    <ImageView
        android:id="@+id/imagenCamara"
        android:layout_width="100dp"
        android:layout_height="80dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        android:contentDescription="imagen camara"
        android:src="@drawable/desafiolatam"
    />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/tituloPregunta"
        android:layout_marginTop="20dp"
        app:layout_constraintTop_toBottomOf="@+id/imagenCamara"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        android:text="Pregunta 1"
        android:textStyle="bold"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/pregunta"
        app:layout_constraintTop_toBottomOf="@+id/imagenCamara"
        app:layout_constraintBottom_toTopOf="@+id/categoria"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"/>

    <TextView
        android:id="@+id/tituloCategoria"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Categoria"
        android:textStyle="bold"
```

```
android:layout_marginTop="30dp"
app:layout_constraintLeft_toLeftOf="parent"
app:layout_constraintRight_toRightOf="parent"
app:layout_constraintTop_toBottomOf="@+id/pregunta"
/>
```

<TextView

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:id="@+id/categoria"
app:layout_constraintTop_toBottomOf="@+id/tituloCategoria"
app:layout_constraintLeft_toLeftOf="parent"
app:layout_constraintRight_toLeftOf="parent"
android:layout_marginTop="20dp"
android:textAlignment="center"/>
```

<RadioGroup

```
android:id="@+id/radioGrupoRespuestas"
android:layout_width="match_parent"
android:layout_height="wrap_content"
app:layout_constraintTop_toBottomOf="@id/categoria"
app:layout_constraintLeft_toLeftOf="parent"
app:layout_constraintRight_toRightOf="parent"
android:layout_marginTop="30dp"
>
```

<RadioButton

```
android:id="@+id/radioRespuestaUno"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
/>
```

<RadioButton

```
android:id="@+id/radioRespuestaDos"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
/>
```

<RadioButton

```
android:id="@+id/radioRespuestaTres"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
/>
```

<RadioButton

```
android:id="@+id/radioRespuestaCuatro"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_gravity="center"
/>
```

</RadioGroup>

<Button

```
android:id="@+id/btnConsultaRespuesta"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginTop="25dp"
```

```

app:layout_constraintTop_toBottomOf="@id/radioGrupoRespuestas"
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintRight_toRightOf="parent"
app:layout_constraintLeft_toLeftOf="parent"
android:text="VER RESPUESTA"
android:background="@color/colorPrimaryDark"
android:textColor="#FFFFFF"
android:paddingHorizontal="20dp"/>

</android.support.constraint.ConstraintLayout>

```

2. Después de haber creado nuestro código layout para la vista, nos ubicamos en nuestra carpeta `cl.desafiolatam.pruebadinamica` y crearemos un nuevo package denominado "fragments".

3. En la carpeta fragments, basados en el código del fragmento en el inicio de este capítulo, vamos a crear una nueva clase java llamada "PreguntaFragment", para la cual se requiere la declaración e inicialización de nuestras vistas del layout `fragment_pregunta.xml`, pasos a seguir:

- Declarar variables de clase:

```

int radioButtonValue = 0;
private TextView preguntaView, categoriaView;
private RadioGroup grupoRespuestasView;
private RadioButton respuestaUno, respuestaDos, respuestaTres, respuestaCuatro;

```

- Declarar un constructor de fragmento (buenas prácticas) con parámetros a recibir desde nuestra actividad en base a nuestra api de datos

```

public static PreguntaFragment newInstance(String pregunta,
                                           String categoria,
                                           String respuestaCorrecta,
                                           ArrayList<String> respuestasIncorrectas) {

    PreguntaFragment fragment = new PreguntaFragment();
    Bundle arguments = new Bundle();
    arguments.putString("PREGUNTA", pregunta);
    arguments.putString("CATEGORIA", categoria);
    arguments.putString("RESPUESTA_CORRECTA", respuestaCorrecta);
    arguments.putStringArrayList("RESPUESTAS_INCORRECTAS", respuestasIncorrectas);
    fragment.setArguments(arguments);

    return fragment;
}

```

- Declaramos el método de ciclo de vida onCreateView() agregando nuestro layout "fragment_pregunta.xml" al contenedor para retornar la vista, declaramos variables de método que capturen los datos recibidos a través de nuestro constructor como argumentos, inicializamos las vistas declaradas anteriormente como variables de clase y asignamos valores a dichas vistas en base a los argumentos recibidos.

```
@Override
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {

    View view = inflater.inflate(R.layout.fragment_pregunta, container, false);

    final String pregunta = Objects.requireNonNull(getArguments()).getString("PREGUNTA");
    final String categoria = Objects.requireNonNull(getArguments()).getString("CATEGORIA");
    final String respuestaCorrecta =
Objects.requireNonNull(getArguments()).getString("RESPUESTA_CORRECTA");
    final ArrayList<String> respuestasIncorrectas = Objects.requireNonNull(getArguments()).getStringArrayList("RESPUESTAS_INCORRECTAS");
    //INICIALIZAMOS LAS VISTAS DECLARADAS
    initializeViews(view);

    //ASIGNANDO VALORES DINAMICOS
    //EN BASE A LOS DATOS RECIBIDOS DE NUESTRA API ASIGNAMOS VALORES A LAS VISTAS
    preguntaView.setText(pregunta);
    categoriaView.setText(categoria);
    //RECORREMOS EL ARREGLO DE STRINGS "RESPUESTAS INCORRECTAS" DE NUESTRA API
    DE DATOS
    for (int x = 0; x < respuestasIncorrectas.size(); x++) {
        switch (x) {
            case 0:
                respuestaUno.setText(respuestasIncorrectas.get(x));
                break;
            case 1:
                respuestaDos.setText(respuestasIncorrectas.get(x));
                break;
            case 2:
                respuestaTres.setText(respuestasIncorrectas.get(x));
                break;
        }
    }
    //AGREGAMOS LA RESPUESTA CORRECTA DE NUESTRA API EN UN CUARTO RADIO BUTTON
    respuestaCuatro = view.findViewById(R.id.radioRespuestaCuatro);
    respuestaCuatro.setText(respuestaCorrecta);

    //EVENTOS DE RADIO BUTTONS - CODIGO PARA QUE LA SELECCION DEL RADIO BUTTON
    SEA ACTUALIZADO EN LA VISTA
    grupoRespuestasView.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener()
    {

        @Override
        public void onCheckedChanged(RadioGroup group, int checkedId) {
            if (respuestaUno.isChecked()) {
                radioButtonValue = 1;
            } else if (respuestaDos.isChecked()) {
                radioButtonValue = 2;
            } else if (respuestaTres.isChecked()) {
```

```

        radioButtonValue = 3;
    } else if (respuestaCuatro.isChecked()) {
        radioButtonValue = 4;
    }
}
});
return view;
}

```

- En el onCreateView() existe un método que inicializa las vistas llamado initializeViews() y el mismo recibe el objeto View que declaramos al inyectar el layout. Este método debemos declararlo para su uso de la siguiente manera:

```

/**
 * Método de inicialización de vistas llamado en el método de ciclo de vida onCreateView
 * @param view Vista fragment_pregunta.xml
 */
private void initializeViews(View view){
    preguntaView = view.findViewById(R.id.pregunta);
    categoriaView = view.findViewById(R.id.categoria);
    grupoRespuestasView = view.findViewById(R.id.radioGrupoRespuestas);
    respuestaUno = view.findViewById(R.id.radioRespuestaUno);
    respuestaDos = view.findViewById(R.id.radioRespuestaDos);
    respuestaTres = view.findViewById(R.id.radioRespuestaTres);
    respuestaCuatro = view.findViewById(R.id.radioRespuestaCuatro);
}

```

4. A continuación editamos nuestro layout "activity_main.xml" para incluir una vista de tipo FrameLayout, la cual cumple la función de ser contenedor de nuestro sublayout del fragmento.

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
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:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MainActivity">

    <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</android.support.constraint.ConstraintLayout>

```

5. Finalmente editamos nuestra clase actividad “MainActivity” específicamente en su método onResponse correspondiente al callback retrofit, quedando de la siguiente manera:

```
@Override
public void onResponse(Call<PreguntasLista> call, Response<PreguntasLista> response) {
    Log.d(TAG, response.toString());
    PreguntasLista preguntas = response.body();

    if (preguntas != null) {
        //MOSTRAMOS PRIMERA PREGUNTA COMO FRAGMENTO PARA VISUALIZAR
        Pregunta pregunta;
        pregunta = preguntas.getResults().get(0);

        PreguntaFragment preguntaFragment = PreguntaFragment
            .newInstance(
                pregunta.getQuestion(),
                pregunta.getCategory(),
                pregunta.getCorrect_answer(),
                pregunta.getIncorrect_answers()
            );

        getSupportFragmentManager().beginTransaction()
            .add(R.id.fragment_container, preguntaFragment, "FRAGMENTO").commit();
    }
}
```

6. Ejecutamos nuestra aplicación para visualizar nuestro primer fragmento “PreguntaFragmento” desplegado en la actividad “MainActivity” y mostrando datos reales de un api de servicios externo a nuestra aplicación, con este resultado ¡hemos dado un gran paso!. Ver imagen 11 de referencia.

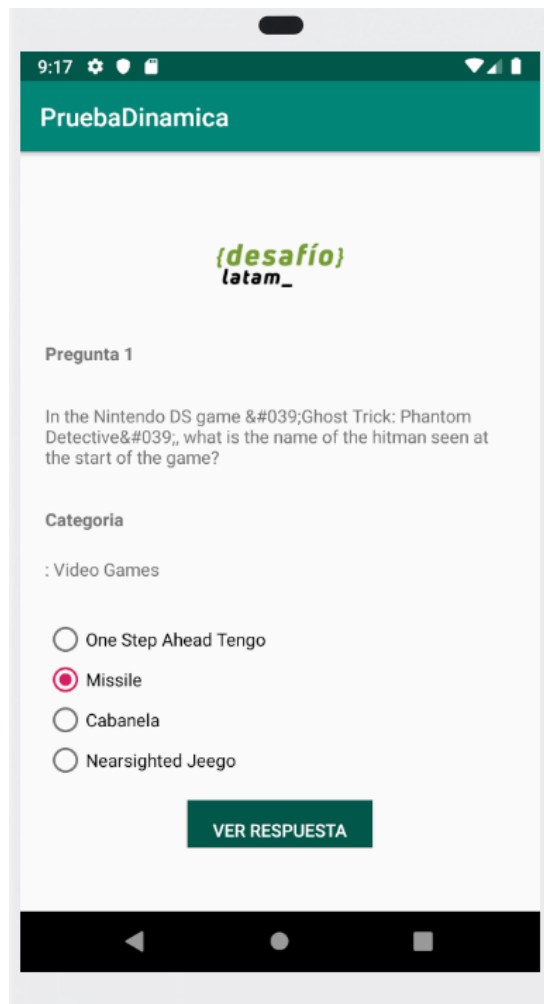


Imagen 11: Primer fragmento

En los próximos capítulos “Fragmentos” e “Integración entre actividades y fragmentos” , seguiremos desarrollando funcionalidades sobre el ciclo de vida de un fragmento y explicaremos en detalle los siguientes conceptos vistos en el ejercicio anterior:

- FragmentManager
- FragmentTransaction
- FrameLayout
- Bundle
- getArguments

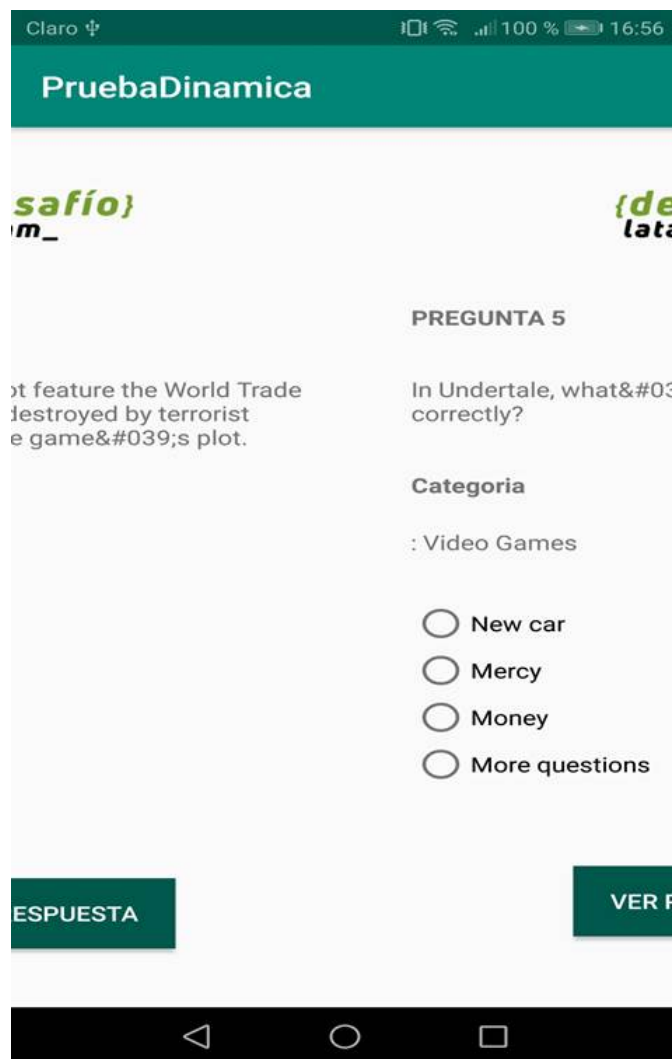


Imagen 12: Vista de dos fragmentos en una actividad

Fragmentos

Competencias:

- Reforzar conceptos de los fragmentos
- Uso del FragmentManager y FragmentTransaction para un fragmento y múltiples fragmentos.

Introducción

Hemos explicado insistentemente durante este curso la importancia de construir aplicaciones android con fragmentos para un óptimo desempeño y un uso limitado de la memoria de los dispositivos, además de ser capaces de reutilizar estos fragmentos en distintas secciones de nuestro código o crear múltiples fragmentos en base a una misma plantilla de diseño.

Los conceptos relacionados con los fragmentos por lo general son preguntas fijas ante cualquier prueba técnica en las búsqueda de trabajo.

Fragmentos

Un fragmento siempre debe estar integrado a una actividad y el ciclo de vida del fragmento se ve directamente afectado por el ciclo de vida de la actividad anfitriona. Por ejemplo, cuando la actividad está pausada, también lo están todos sus fragmentos, y cuando la actividad se destruye, lo mismo ocurre con todos los fragmentos.

Cuando una actividad se está ejecutando, puedes manipular cada fragmento de forma independiente; por ejemplo, para agregarlos o quitarlos. La imagen 12 muestra dos fragmentos exhibidos en la misma actividad en diferentes tamaños de pantalla. En el caso de una pantalla grande, ambos fragmentos pueden ubicarse uno al lado del otro y mostrarse al mismo tiempo, pero en un teléfono celular, dadas sus dimensiones, sólo será posible visualizar un fragmento a la vez, de modo que los fragmentos se reemplazan unos a otros a medida que el usuario navega a través de nuestra aplicación.



Imagen 13: Fragmentos en diferentes tamaños de pantallas

En el ejemplo del capítulo anterior creamos un fragmento de nombre "FragmentoPregunta.java" en el cual declaramos un constructor para definir su comportamiento inicial y recibir datos de entrada, además de declarar, inicializar y asignar datos de las variables asociadas a nuestro layout del fragmento llamado "fragment_pregunta.xml". Estas dos acciones junto con la declaración e instancia del fragmento, fragment manager y fragment transaction desde la actividad son necesarias para el despliegue efectivo de nuestro fragmento en la aplicación.

Agregando un fragmento a una actividad

Cuando agregas un fragmento como parte del layout de tu actividad, este se ubica en un ViewGroup, dentro de la jerarquía de vistas de la actividad y el fragmento define su propio layout de vista, como lo hicimos con el caso del layout "fragment_pregunta.xml".

Para poder agregar un fragmento a una actividad, y esta a su vez al hilo principal de la interfaz de usuario android (IU), existen dos maneras:

- Declarar el fragmento en el archivo de layout de la actividad.

En este caso, puedes especificar propiedades de diseño para el fragmento como si fueran una vista. Por ejemplo, incorporando un :

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
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:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MainActivity">

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

</android.support.constraint.ConstraintLayout>
```

o también con un :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:name="com.desafiolatam.fragments.ArticleListFragment"
        android:id="@+id/list"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
    <fragment android:name="com.desafiolatam.fragments.ArticleReaderFragment"
        android:id="@+id/viewer"
        android:layout_weight="2"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
</LinearLayout>
```

- La otra forma es agregar el fragmento de manera programática (en el código) en un ViewGroup ya existente.

En cualquier momento mientras tu actividad se esté ejecutando, puedes agregar fragmentos al layout de una actividad. Solo necesitas especificar un ViewGroup en el cual se inyectará el fragmento.

Para realizar transacciones de fragmentos en tu actividad (como agregar, quitar o reemplazar un fragmento), debes usar las API de FragmentTransaction. Puedes obtener una instancia de FragmentTransaction de tu Activity como lo muestra el siguiente ejemplo:

```
FragmentManager fragmentManager = getFragmentManager();
FragmentTransaction fragmentTransaction =
    fragmentManager.beginTransaction();
```

Luego puedes agregar un fragmento usando el método add() y especificando el fragmento a agregar y la vista en la que se insertará. Por ejemplo:

```
ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```

El primer argumento que se pasa al método add(), es un ViewGroup, en el que se debe colocar el fragmento especificado por su identificador (id), y el segundo parámetro es el fragmento (la instancia) que quieres agregar.

Una vez que realices los cambios con, debes llamar al método commit() para que se apliquen esos cambios.

Fragment Manager

La clase `FragmentManager` proporciona métodos que te permiten agregar, quitar y reemplazar fragmentos en una actividad durante el tiempo de ejecución a fin de crear una experiencia dinámica para el usuario de la aplicación.

Fragment Transaction

Para realizar una transacción, como agregar o quitar un fragmento, debes usar **FragmentManager** para crear una `FragmentTransaction`, que proporciona el api de android para agregar, quitar, reemplazar y realizar otras transacciones de fragmentos.

Si tu actividad permite quitar y reemplazar fragmentos, debes agregar los fragmentos principales a la actividad durante el método `onCreate()` de la actividad.

Una regla importante al trabajar con fragmentos, especialmente al agregar fragmentos durante el tiempo de ejecución de nuestra aplicación, es que el layout de tu actividad debe incluir una vista en la cual se pueda inyectar nuestro fragmento como lo hemos indicado anteriormente.

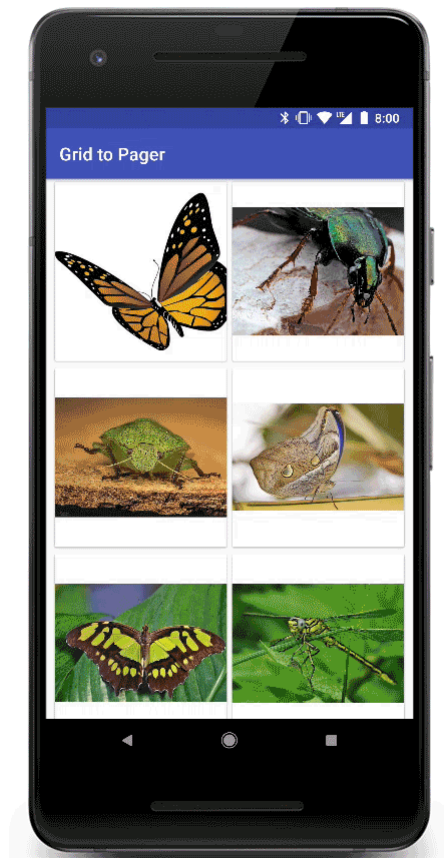


Imagen 14: Fragment Transaction

En el ejercicio 5 del capítulo "Ciclo de vida de un fragmento" utilizamos las siguientes instrucciones al uso del fragment manager y fragment transaction:

```
getSupportFragmentManager().beginTransaction()  
    .add(R.id.fragment_container, preguntaFragment, "FRAGMENTO").commit();
```

En esta instrucción invocamos el método `getSupportFragmentManager` de `FragmentManager` y el `.beginTransaction()` de `FragmentTransaction`, para agregar el fragmento "preguntaFragment" a la vista `FrameLayout` de `id fragment_container`, declarada en nuestro `layout` de actividad "activity_main.xml".

Reemplazar un Fragmento

El procedimiento para reemplazar un fragmento es similar al que se sigue para agregar un fragmento, pero requiere del uso del método `replace()` en lugar de `add()`.

Cuando realizamos transacciones entre fragmentos, como reemplazar o agregar, se recomienda permitirle al usuario retroceder en la navegación y deshacer el cambio. Para permitirle al usuario retroceder en la navegación de las transacciones con fragmentos, debes invocar a `addToBackStack()` antes de confirmar la `FragmentTransaction`. Un ejemplo de reemplazar un fragmento que ya se encuentre en nuestro `ViewGroup` "fragment_container" sería el siguiente:

```
PreguntaFragment fragmento = new PreguntaFragment();
Bundle args = new Bundle();
args.putInt(PreguntaFragment.ARG_POSITION, position);
fragmento.setArguments(args);

FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();

// Reemplazamos lo que exista en el ViewGroup fragment_container
transaction.replace(R.id.fragment_container, fragmento);
transaction.addToBackStack(null);

// Commit transaction
transaction.commit();
```

Lista de Fragmentos a partir de un mismo layout

En concordancia con nuestro proyecto "PruebaDinámica" es necesario explicar la práctica avanzada de crear múltiples fragmentos a partir de un arreglo de datos recibidos desde un api, utilizando una misma clase de fragmento y un mismo `layout`, invocando nuevas instancias del fragmento y guardando estos en una lista que luego será iniciada a través del `fragment manager`.

Entendiendo que nuestra api trae el objeto tipo arreglo "results" con hasta diez preguntas en su interior, crearemos un total de 10 fragmentos basados en esta lista, cada uno representando una pregunta con sus datos respectivos. Para lograr este comportamiento es necesario seguir los siguientes pasos:

- Crear un método privado de clase en la actividad que devuelva una lista de fragmentos a partir de la iteración de nuestra lista "results". Ejemplo:

```
private List<Fragment> getFragmentosPag(List<PreguntasLista>
listPreguntas, Bundle datosPreguntas){}
```

- Crear una pequeña clase que extienda de la librería de soporte FragmentPagerAdapter declarando un método constructor que reciba una instancia de FragmentManager y nuestra lista anterior de fragmentos, además de declarar los métodos propios del adapter getCount() y getItem().

```
public class PruebaPagerAdapter extends FragmentPagerAdapter{
```

- Declarar una variable de la clase ViewPager que reciba la instancia de nuestro adaptador. Por ejemplo:

```
List<Fragment> paginasFragmentos = getFragmentosPag(listPreguntas, datosPreguntas);

pruebaPagerAdapter = new PruebaPagerAdapter(getSupportFragmentManager(), paginasFragmentos);

viewPager = (ViewPager) findViewById(R.id.viewpager);
viewPager.setAdapter(pruebaPagerAdapter);
```