

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONESMULTIPLATAFORMA

Servicios en red
codificación JSON

XAMPP

PHP: lenguaje de programación backend

Veremos una manera de salvaguardar la base de datos de una app aunque ésta se desinstale del dispositivo móvil.

Introducción a JSON

Cuando queremos que una **aplicación**, independientemente de que sea de escritorio o de móvil, pueda **comunicarse con otra** que está en otro sistema, necesitaremos un **estándar** para que todas las aplicaciones puedan entenderlo.

En nuestro caso, vamos a comunicar una **aplicación móvil con un servidor** remoto y, para ello, podremos utilizar **dos tipos de estándares, XML y JSON**.

JSON es una **codificación de texto** que sirve para **intercambiar datos** de una forma **sencilla y estructurada**. Este tipo de formato presenta una serie de **ventajas sobre el XML**, que ya estudiamos en otras asignaturas, y es que **JSON es muy fácil de evaluar**, aunque esto no significa que XML no se utilice, todo lo contrario: en una **misma aplicación, se pueden utilizar tanto JSON como XML** para realizar las comunicaciones.

JSON soporta los siguientes datos:

- **Números:** Se pueden representar números positivos, negativos, enteros y reales, **separados por puntos**. Por ejemplo: 123.456
- **Cadenas:** Representan secuencias de **cero o más caracteres**. Se ponen entre **comillas dobles**. Por ejemplo: "Hola".
- **Booleanos:** Representan valores booleanos y pueden tener dos valores: true y false.
- **Null:** Representan el valor nulo.
- **Array:** Representa una **lista ordenada** de **cero o más valores**, los cuales pueden ser de **cualquier tipo**. Los valores se separan por comas y el **vector** se introduce entre **corchetes**. Por ejemplo ["valor 1", "valor 2", "valor 3"].
- **Objetos:** Este tipo de dato viene representado por una **colección que no está ordenada**, compuesta por elementos del tipo <nombre>:<valor>, que podremos **separar por comas** y **delimitar entre llaves**. El **nombre**, obligatoriamente, debe ser una **cadena**, mientras que el **valor** puede ser **cualquier tipo** de dato conocido.

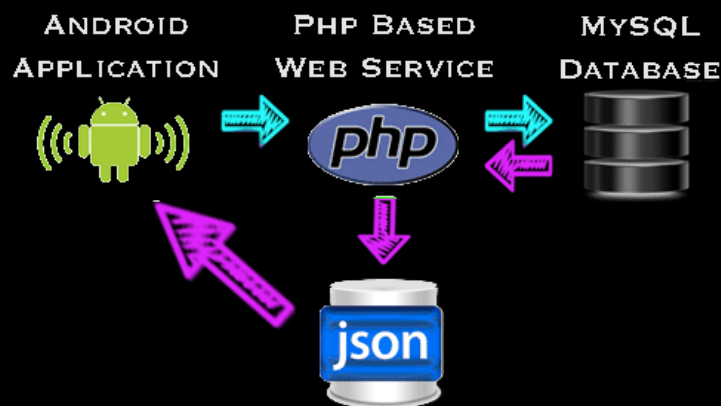
Ejemplo de JSON simple:

```
{
  "departamento": 8,
  "nombredepto": "Ventas",
  "director": "Juan Rodríguez",
  "empleados": [
    {
      "nombre": "Pedro",
      "apellido": "Fernández"
    },
    {
      "nombre": "Jacinto",
      "apellido": "Benavente"
    }
  ]
}
```

Usando JSON en Android

Vamos a conectar las aplicaciones Android con un servidor remoto mediante este estándar. Es decir, nuestra aplicación se conectará a un servidor externo que use **MySQL** para almacenar los datos, el **servidor obtendrá los datos** que se pidan y los **moldeará en formato JSON**, devolviéndolos a nuestra aplicación. Esta los **leerá**, pero necesitaremos un «intermediario» para realizar este proceso. Concretamente, este intermediario será **PHP**.

Esquema Android - PHP - JSON - MySQL:



Para poder tratar con los datos en JSON, deberemos usar las clases:

JSONObject, que almacenará un objeto con formato JSON;

JSONArray, que guardará un array de objetos JSONObject. Estas clases vienen directamente integradas en la **JDK de Android**, así que no es necesario agregar ninguna biblioteca externa.

Un **JSONObject** o un **JSONArray** pueden ser creados directamente desde un **String que cumpla el formato JSON**, utilizando sus respectivos **constructores**.

Cada vez que realicemos operaciones con JSON deberemos utilizar la **excepción JSONException**.

Los JSONArray son un array **equivalente a un ArrayList**, salvo que ya está optimizado para datos del tipo JSON, por lo que, en los JSONArray, vamos a tener todos los **métodos** de los **ArrayList**.

En los objetos de tipo JSON, tendremos los **métodos get** para cada tipo de dato, getString, getInt, etc., que nos devolverán su valor, **pasándole el nombre del dato**.

Ejemplo de JSON paso a paso:

En el fichero MainActivity

```
public class MainActivity extends AppCompatActivity {

    private RecyclerView listaPescados;
    private AdaptadorPescado adaptador;

    private final String CONTENIDO = "[{\"fish_name\":\"Indian Mackerel\",\"cat_name\":\"Marine Fish\",\"size_name\":\"Medium\",\"price\":\"100\"},\n" +
        "{\"fish_name\":\"Manthal Repti\",\"cat_name\":\"Marine Fish\",\"size_name\":\"Small\",\"price\":\"200\"},\n" +
        "{\"fish_name\":\"Baby Sole Fish\",\"cat_name\":\"Marine Fish\",\"size_name\":\"Small\",\"price\":\"600\"},\n" +
        "{\"fish_name\":\"Silver Pomfret\",\"cat_name\":\"Marine Fish\",\"size_name\":\"Large\",\"price\":\"300\"},\n" +
        "{\"fish_name\":\"Squid\",\"cat_name\":\"Shell Fish\",\"size_name\":\"Small\",\"price\":\"800\"},\n" +
        "{\"fish_name\":\"Clam Meat\",\"cat_name\":\"Shell Fish\",\"size_name\":\"Small\",\"price\":\"350\"},\n" +
        "{\"fish_name\":\"Indian Prawns\",\"cat_name\":\"Shell Fish\",\"size_name\":\"Medium\",\"price\":\"270\"},\n" +
        "{\"fish_name\":\"Mud Crab\",\"cat_name\":\"Shell Fish\",\"size_name\":\"Medium\",\"price\":\"490\"},\n" +
        "{\"fish_name\":\"Grey Mullet\",\"cat_name\":\"Backwater Fish\",\"size_name\":\"Small\",\"price\":\"670\"},\n" +
```

```

        "{\"fish_name\":\"Baasa\",\"cat_name\":\"Backwater
Fish\",\"size_name\":\"Large\",\"price\":\"230\"},\n" +
        "{\"fish_name\":\"Pearl Spot\",\"cat_name\":\"Backwater
Fish\",\"size_name\":\"Small\",\"price\":\"340\"},\n" +
        "{\"fish_name\":\"Anchovy\",\"cat_name\":\"Marine
Fish\",\"size_name\":\"Small\",\"price\":\"130\"},\n" +
        "{\"fish_name\":\"Sole Fish\",\"cat_name\":\"Marine
Fish\",\"size_name\":\"Medium\",\"price\":\"250\"},\n" +
        "{\"fish_name\":\"Silver Croaker\",\"cat_name\":\"Marine
Fish\",\"size_name\":\"Small\",\"price\":\"220\"}]]";

```

@Override

```

protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // ***** Ligamos los elementos de la actividad *****
    listaPescados = findViewById(R.id.listaPescados);
    // *****
    try
    {
        ArrayList<Pescado> datospescados = new ArrayList<>();
        // Creo el JSONArray con los datos en formato JSON
        JSONArray jArray = new JSONArray(CONTENIDO);
        // Extraigo los datos que están en JSON
        for (int i = 0; i < jArray.length(); i++)
        {
            JSONObject json_data = jArray.getJSONObject(i);
            String nombre = json_data.getString("fish_name");
            String categoria = json_data.getString("cat_name");
            String tamano = json_data.getString("size_name");
            int precio = json_data.getInt("price");
            Pescado pescado = new Pescado(nombre, categoria, tamano, precio);
            datospescados.add(pescado);
        }
        // Mostramos la lista
        // Con esto el tamaño del recyclerview no cambiará
        listaPescados.setHasFixedSize(true);
        // Creo un layoutManager para el recyclerview
        LinearLayoutManager llm = new LinearLayoutManager(this);
        listaPescados.setLayoutManager(llm);
        adaptador = new AdaptadorPescado(this, datospescados);
        listaPescados.setAdapter(adaptador);
        adaptador.refrescar();
    }
}

```

```
    catch (JSONException e)
    {
        System.out.println("Error: " + e.toString());
    }
}
}
```

En modelo (clase **Pescado**)

```
public class Pescado {

    private String nombre;
    private String categoria;
    private String tamano;
    private int precio;

    public Pescado(String nombre, String categoria, String tamano, int precio)
    {
        this.nombre = nombre;
        this.categoria = categoria;
        this.tamano = tamano;
        this.precio = precio;
    }

    public String getNombre()
    {
        return nombre;
    }

    public void setNombre(String nombre)
    {
        this.nombre = nombre;
    }

    public String getCategoria()
    {
        return categoria;
    }

    public void setCategoria(String categoria)
    {
        this.categoria = categoria;
    }
}
```

```

public String getTamano()
{
    return tamano;
}

public void setTamano(String tamano)
{
    this.tamano = tamano;
}

public int getPrecio()
{
    return precio;
}

public void setPrecio(int precio)
{
    this.precio = precio;
}
}

```

Y el AdaptadorPescado, para crear el RecyclerView

```

public class AdaptadorPescado extends
RecyclerView.Adapter<AdaptadorPescado.HolderPescado>{

    public static class HolderPescado extends RecyclerView.ViewHolder
    {
        ImageView ivFish;
        TextView textFishName, textPrice, textSize, textType;

        HolderPescado(View itemView)
        {
            super(itemView);
            ivFish = itemView.findViewById(R.id.ivFish);
            textFishName = itemView.findViewById(R.id.textFishName);
            textPrice = itemView.findViewById(R.id.textPrice);
            textSize = itemView.findViewById(R.id.textSize);
            textType = itemView.findViewById(R.id.textType);
        }
    };
}

```

```

private ArrayList<Pescado> datos;
private Context contexto;

public AdaptadorPescado(Context contexto, ArrayList<Pescado> datos)
{
    this.contexto = contexto;
    this.datos = datos;
}

/**
 * Agrega los datos que queremos mostrar
 * @param datos Datos a mostrar
 */
public void add(ArrayList<HolderPescado> datos)
{
    datos.clear();
    datos.addAll(datos);
}

/**
 * Actualiza los datos del RecyclerView
 */
public void refrescar()
{
    notifyDataSetChanged();
}

@Override
public HolderPescado onCreateViewHolder(ViewGroup parent, int viewType)
{
    View v = LayoutInflater.from(parent.getContext()).inflate(R.layout.elemento_pescado,
parent, false);
    HolderPescado pvh = new HolderPescado(v);
    return pvh;
}

@Override
public void onBindViewHolder(HolderPescado pescadoactual, int position)
{
    pescadoactual.ivFish.setImageResource(R.drawable.ic_img_fish);
    pescadoactual.textFishName.setText(datos.get(position).getNombre());
    pescadoactual.textPrice.setText(String.valueOf(datos.get(position).getPrecio()));
    pescadoactual.textSize.setText(datos.get(position).getTamano());
    pescadoactual.textType.setText(datos.get(position).getCategoria());
    pescadoactual.textPrice.setTextColor(ContextCompat.getColor(contexto,

```



```
R.color.colorAccent));  
    }  
  
    @Override  
    public int getItemCount()  
    {  
        return datos.size();  
    }  
  
    @Override  
    public void onAttachedToRecyclerView(RecyclerView recyclerView)  
    {  
        super.onAttachedToRecyclerView(recyclerView);  
    }  
}
```

Los costes en los servidores remotos

Existen ciertos proveedores servidores remotos que ofrecen packs muy básicos de forma gratuita, pero, como es lógico, estos packs van a estar **muy limitados**.

Estas limitaciones se basan en el **uso de espacio para nuestro hosting**, es decir, el espacio en disco que podremos usar en el servidor remoto; también estarán limitadas el **número de bases de datos** que podremos gestionar, y, por último, habrá momentos en los que los **servidores** con servicio gratuito **no estarán disponibles**.

La suma de todo esto hará que nuestras aplicaciones no funcionen de un modo normal ni fiable.

Para el **uso didáctico**, este tipo de servidores **nos van a servir** a la perfección, ya que, normalmente, se usan para pruebas en las que estaremos continuamente montando y desmontando servicios para nuestro aprendizaje, sin ningún otro objetivo, es decir, sin dar un servicio a terceros (clientes por ejemplo). Por tanto, para esto sí son una opción que podemos tener muy en cuenta, y no tendremos que hacer un desembolso económico, por pequeño que sea.

Un ejemplo de este tipo de servicios gratuitos lo podemos encontrar en esta compañía: **<https://es.000webhost.com/>**

Introducción a PHP7

Para poder acceder y gestionar la base de datos del servidor externo que utilizaremos, necesitaremos un lenguaje de programación que se pueda ejecutar en la **parte de servidor**, conocida como **backend**.

Existen varios **lenguajes adecuados** para esto como, por ejemplo, **Python**, pero, en nuestro caso, vamos a utilizar **PHP** en su **versión 7**, ya que **soporta** la **programación dirigida a objetos**.

PHP es un **lenguaje de programación web** que se puede integrar fácilmente en ficheros HTML.

El intérprete de PHP solo **ejecutará el código** que se encuentra **entre sus delimitadores**, los cuales son:

<?php para abrir una sección PHP

?> para cerrarla. *

* Aunque es una práctica común en archivos que contienen solo código PHP omitir la etiqueta de cierre **?>** para evitar problemas de espacios en blanco.

El propósito de estos delimitadores es **separar el código PHP** del resto de código.

Las **variables** en este lenguaje de programación se van a **prefijar** con el **símbolo del dólar (\$)**, y **no es necesario indicarle el tipo**, ya que lo adoptarán **automáticamente** cuando se igualen a algo.

Este lenguaje de programación soporta instrucciones **condicionales, bucles, funciones y clases**, como otros lenguajes de programación. Todo esto lo veremos poco a poco de una forma muy superficial en esta asignatura, únicamente con el objetivo de gestionar una base de datos.

Es importante no olvidar que, al igual que en Java, todas las instrucciones deberán **acabar en punto y coma (;)**.

En el bloque que estudiaremos sobre PHP, nos vamos a centrar en su funcionamiento como **mediador con la base de datos**. Así, no necesitaremos introducir datos por el usuario, pero sí **mostrar datos por pantalla** y, al ser un lenguaje de programación web, se **mostrarán en un navegador** (Firefox, Chrome, etc.), lo cual podremos hacer mediante la **instrucción echo**. Esta instrucción mostrará, en el navegador que utilicemos, el mensaje que queramos, pudiendo mostrar el valor de variables, como ya sabemos hacer.

Hola mundo en PHP. Guardado con extensión .php:

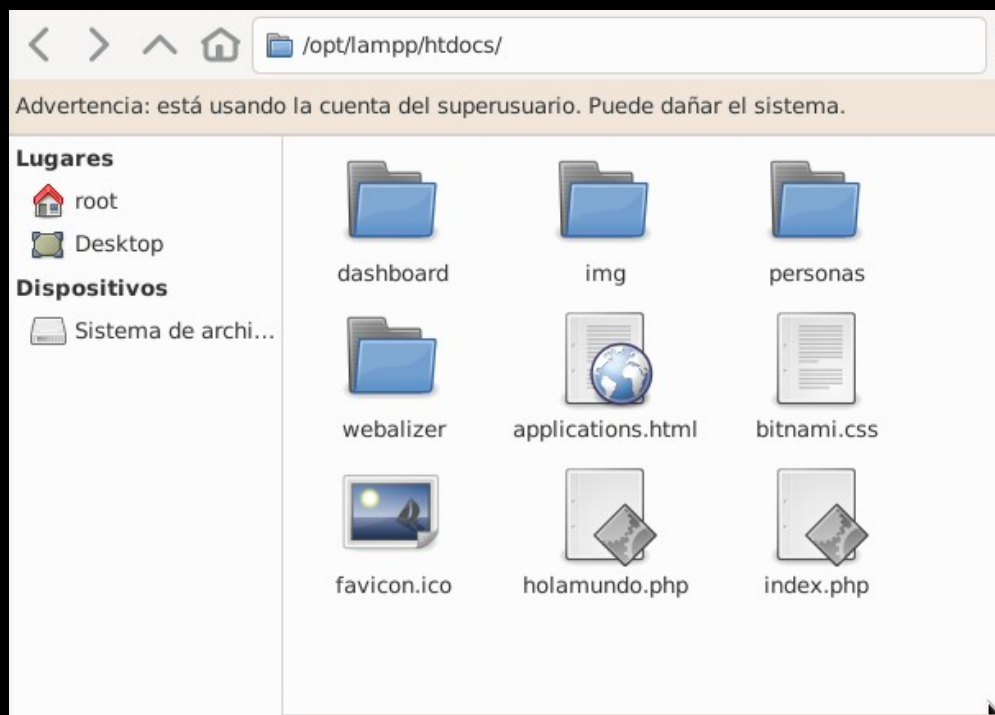
```
<!DOCTYPE HTML>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <title>Ejemplo básico PHP</title>
  </head>
```

```
<body>
  <p>
    <?php
      echo "Hola Mundo";
    ?>
  </p>
</body>
</html>
```

Si queremos ejecutar este archivo en nuestro navegador, se ejecutaría sin necesidad de incluirlo en un elemento html, de hecho esto se ejecutaría:

```
<?php
echo "Hola Mundo";
?>
```

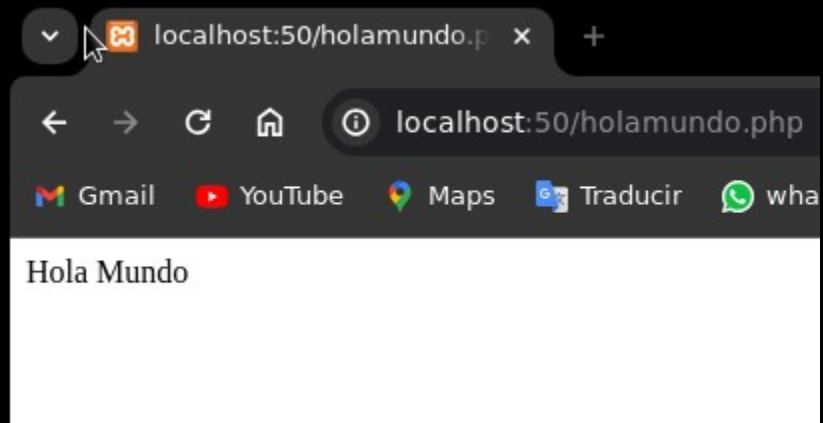
Pues bien, vamos al directorio donde instalamos nuestro XAMPP (LAMPP en Linux. La instalación de XAMPP la veremos más adelante en este pdf). Y buscamos el directorio htdocs, como se muestra en la siguiente captura, y ahí, con permisos de administrador (o de superusuario en Linux), incluimos nuestro archivo holamundo.php con el código de arriba, cualquiera de los dos.



Despues de lo cual, con nuestro servidor Apache “running” con XAMPP, vamos a nuestro navegador y escribimos como url a buscar:

http://localhost:50/holamundo.php

Si es el caso de que el puerto de nuestro servidor es el 50, si no lo sustituimos por el nuestro. Y veremos nuestro hola mundo en nuestro navegador:



Tipos de servidores

Como hemos comentado existen dos tipos de servidores remotos, los gratuitos, como un peor servicio, y los servidores de pago, que, aunque tengan un mejor servicio implican un desembolso económico.

*Si no queremos dejarnos un dinero y tampoco queremos que nuestro servidor de problemas, una opción a tener en cuenta para tener un servidor que nos permita practicar con lo que estamos aprendiendo, es **utilizar un servidor local**.*

*Esto lo podemos conseguir con programas como **XAMPP** por ejemplo, sobre el que profundizaremos a continuación.*

*En la web **apachefriends.org** puedes descargarlo y obtener más información.*

Instalación y configuración de PHP7 y MYSQL

Si no queremos utilizar un servidor web para utilizar PHP junto a MySQL, podremos instalarlos en nuestro propio equipo, convirtiéndolo en un servidor local, que funcionará a la perfección. De esta forma, cualquier contratiempo podrá resolverse sin depender de terceros.

Para ello, necesitaremos instalar XAMPP, que podremos descargar de su página web: <https://www.apachefriends.org/es/download.html>

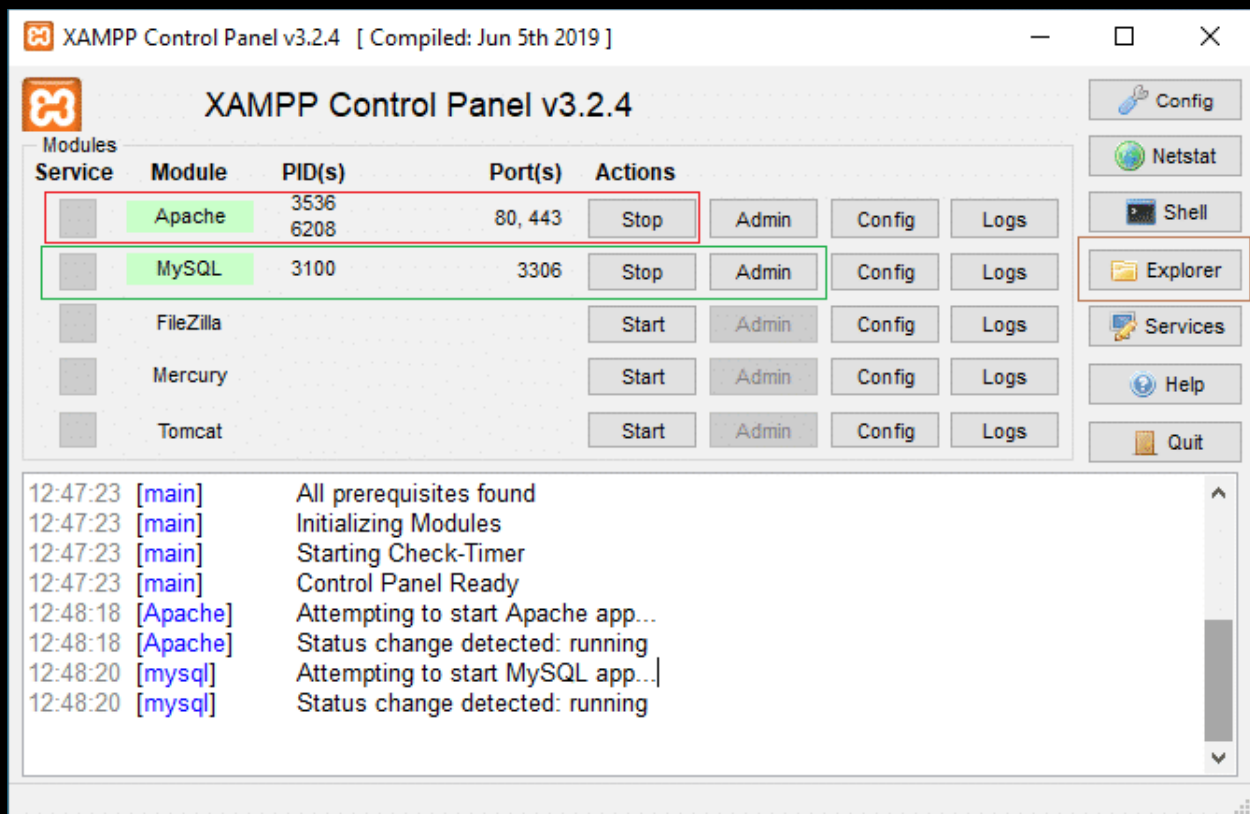
XAMPP nos ofrece, además de un **gestor de bases de datos MySQL**, la posibilidad de utilizar un **intérprete de PHP**.

Es decir, se **instalará un pequeño servidor** en nuestro ordenador, que podremos arrancar siempre que lo necesitemos, pudiendo utilizar, por tanto, PHP y MySQL para probar nuestras aplicaciones Android.

Una vez tengamos instalado XAMPP, tendremos las siguientes opciones disponibles.

Marcado en el **cuadrado rojo** tenemos la **opción de Apache**, que necesitamos para que nuestro **servidor** esté operativo. Pulsaremos el botón **Iniciar**.

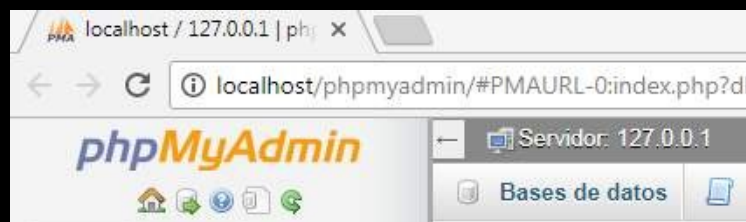
En el **recuadro verde** tenemos la **opción de MySQL**, que es necesario que activemos para que nuestra **base de datos** esté operativa. Pulsaremos **Iniciar**.



Si queremos iniciar el **gestor de bases de datos**, en este caso **phpMyAdmin**, deberemos pulsar en el **botón Admin** de la parte de **MySQL**, lo cual hará que se **abra** desde nuestro **navegador** predeterminado. El **usuario**, por defecto, es **root** y su **contraseña** está **vacía**.

En Linux (XAMPP es llamado LAMPP en la instalación), para acceder a **phpMyAdmin** no tendremos el botón Admin, de modo que una vez estén levantados el FTP, el servidor Apache y la BBDD MySQL, iremos a nuestro navegador y escribiremos como URL: <http://localhost:50/phpmyadmin/>, donde "50" es el puerto que le hemos asignado, que normalmente será el 80 si no lo tenéis ocupado con otro servicio en vuestro sistema operativo. Si tenéis dudas, podéis comprobar qué puerto tiene asignado XAMPP a Apache (servidor) si pulsamos en el botón "Configure" teniendo seleccionado Apache Web Server.

phpMyAdmin en localhost:



Para utilizar nuestro servidor local, deberemos hacer referencia a **localhost** en nuestro navegador o, en su defecto, a la dirección IP **127.0.0.1**.

Si no se conecta el servidor local de Apache, lo más probable será que sea porque el puerto usado esté ocupado por otro servicio. De modo que lo solucionaremos cambiando el puerto, pulsando en el botón “config” podremos cambiar el puerto. Si el 80 el puerto que da problemas, podríamos poner el 8080, o el 50, si no están ocupados por otro servicio. Este último me dió buen resultado en Debian (Linux).

Creación de base de datos Personas y de la tabla Persona:

Abrimos nuestro [phpmyadmin](#), y seleccionando la pestaña SQL vamos ejecutando cada sentencia SQL para la creación de la tabla:

```
CREATE DATABASE personas;  
USE personas;
```

para la creación de la tabla persona:

```
CREATE TABLE IF NOT EXISTS persona (  
  DNI VARCHAR(10) PRIMARY KEY,  
  nombre VARCHAR(100) NOT NULL,  
  apellidos VARCHAR(100) NOT NULL,  
  telefono int NOT NULL,  
  email VARCHAR(100) NOT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

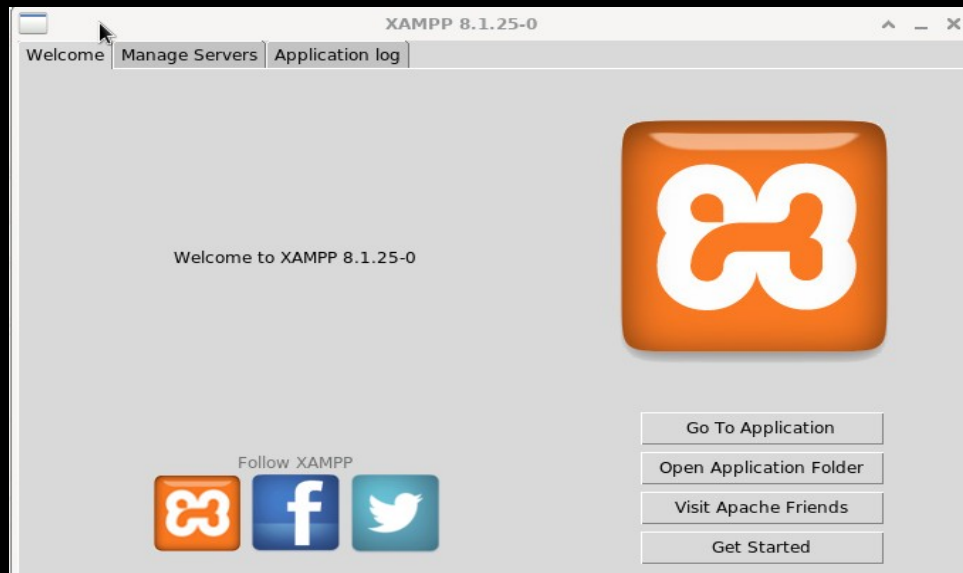
y para la inserción de registros nuevos en la tabla personas:

```
INSERT INTO persona (DNI, nombre, apellidos, telefono, email) VALUES ('14786325P',  
'Juan', 'López López', 666666666, 'juanlopez@gmail.com');  
INSERT INTO persona (DNI, nombre, apellidos, telefono, email) VALUES ('24158749L',  
'Pepe', 'Adamuz Núñez', 666956666, 'pepeadamuz@gmail.com');  
INSERT INTO persona (DNI, nombre, apellidos, telefono, email) VALUES ('36214758I',  
'Dolores', 'Pérez Aguilera', 684766666, 'doloresperez@gmail.com');
```

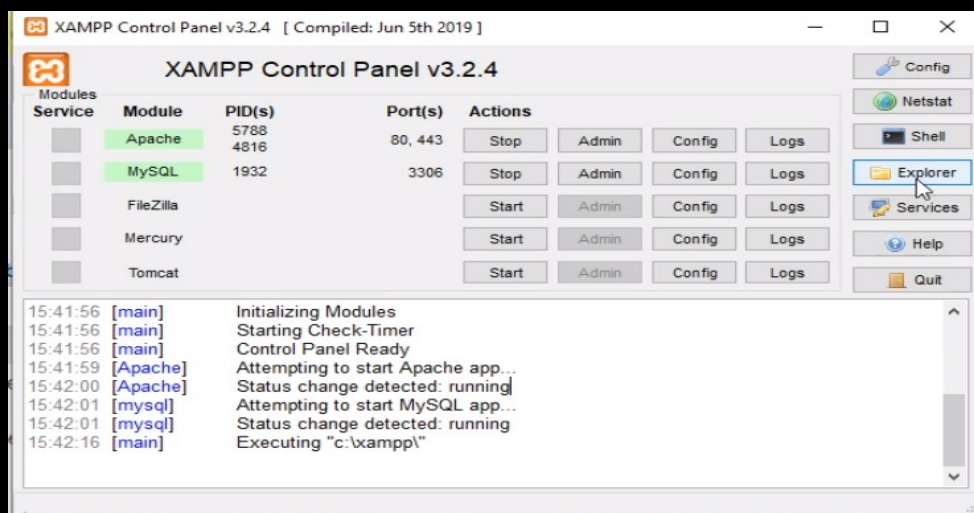
Pulsando en el botón “continuar” se irá ejecutando cada petición al gestor de BBDD MySQL...

Los ficheros php deberemos de colocarlos dentro de nuestro servidor Apache. Para ello iremos al botón “exporer” si estamos en XAMPP de windows, o a “open application folder” si estamos en XAMPP de Linux (LAMPP) y se nos abrirá el explorador en el directorio de XAMPP:

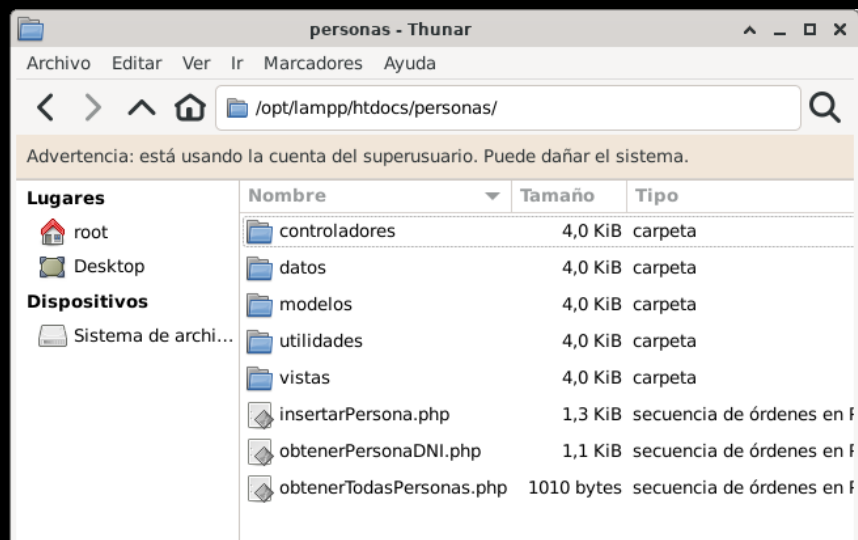
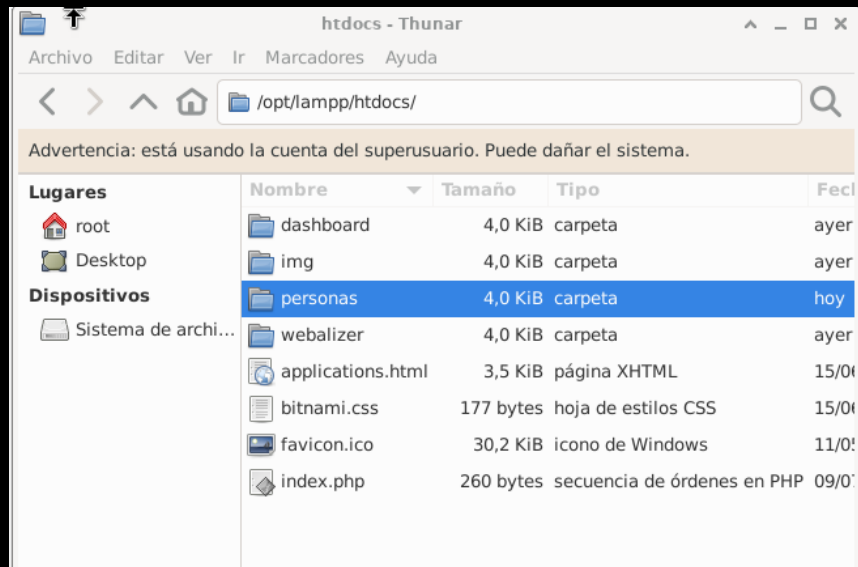
Vista de XAMPP en Linux ("Open Application Folder" para poder agregar nuestros archivos .php)



Vista de XAMPP en Windows ("Explorer" para poder agregar nuestros archivos .php)



Y nos situaremos en la carpeta /htdocs, donde agregaremos un directorio con el modelo vista controlador visto anteriormente:



Este directorio personas estará subido como directorio en este mismo repositorio de github.

Dentro del directorio modelos, tenemos a la clase Persona.php:

```
<?php
```

```
// Esta clase representa una persona
```

```
class Persona
```

```
{
```

```
    // Variables de clase
```

```

private $DNI, $nombre, $apellidos, $telefono, $email;

// Constructor
public function __construct($nDNI, $nnombre, $napellidos, $ntelefono, $nemail)
{
    $this->DNI = $nDNI;
    $this->nombre = $nnombre;
    $this->apellidos = $napellidos;
    $this->telefono = $ntelefono;
    $this->email = $nemail;
}

// Muestra los datos de la persona
public function toString()
{
    return
        [
            "DNI" => utf8_encode($this->DNI),
            "nombre" => utf8_encode($this->nombre),
            "apellidos" => utf8_encode($this->apellidos),
            "telefono" => utf8_encode($this->telefono),
            "email" => utf8_encode($this->email)
        ];
}
}

```

Dentro del directorio controladores tenemos al controladorPersona.php:

```

<?php
require_once('datos/ConexionBD.php');
require_once('utilidades/ExcepcionApi.php');
require_once('datos/mensajes.php');
// Esta clase representa un controlador para las personas
class ControladorPersonas
{
    // Nombres de la tabla y de los atributos, aunque no es obligatorio nos facilitará mucho todo
    const NOMBRE_TABLA = "persona";
    const DNI = "DNI";
    const NOMBRE = "nombre";
    const APELLIDOS = "apellidos";
    const TELEFONO = "telefono";
    const EMAIL = "email";
    /**

```

```

* Descripción: Obtiene los datos de todas las personas que hay en el sistema
* mediante un SELECT *
* Este método se corresponderá con el fichero que está en la raíz del proyecto llamado
* obtenerTodasPersonas.
* @return Datos de todas las personas que hay en el sistema
*/

```

```

public function obtenerTodasPersonas()
{
    try
    {
        // Conectará a la base de datos con esta instrucción:
        $pdo = ConexionBD::obtenerInstancia()->obtenerBD();

        // Sentencia SELECT
        $comando = "SELECT * FROM " . self::NOMBRE_TABLA;

        $sentencia = $pdo->prepare($comando);

        // Dentro de $sentencia tendremos todos los resultados de la consulta
        $sentencia->execute();

        $array = array();

        // El método fetch nos irá devolviendo filas completas:
        while ($row = $sentencia->fetch(PDO::FETCH_ASSOC))
        {
            // Iremos agregando las filas a $array con el método array_push:
            array_push($array, $row);
        }

        return [
            [
                "estado" => ESTADO_CREACION_EXITOSA,
                "mensaje" => $array
            ]
        ];
    }
    catch (PDOException $e)
    {
        throw new ExcepcionApi(ESTADO_ERROR_BD, $e->getMessage());
    }
}
/**

```

```

* Descripción: Obtiene y devuelve una persona según su DNI
* @param DNI DNI de la persona

```

```

* @return Datos de la persona indicada con su DNI
*/
public function obtenerPersonaDNI($DNI)
{
    try
    {
        $pdo = ConexionBD::obtenerInstancia()->obtenerBD();

        // Sentencia SELECT
        $comando = "SELECT * FROM " . self::NOMBRE_TABLA . " " .
            "WHERE " . self::DNI . " = ?";

        $sentencia = $pdo->prepare($comando);

        // Pongo los datos en la consulta INSERT
        $sentencia->bindParam(1, $DNI);

        $sentencia->execute();

        $array = array();

        while ($row = $sentencia->fetch(PDO::FETCH_ASSOC))
        {
            array_push($array, $row);
        }

        return [
            [
                "estado" => ESTADO_CREACION_EXITOSA,
                "mensaje" => $array
            ]
        ];
    }
    catch (PDOException $e)
    {
        throw new ExcepcionApi(ESTADO_ERROR_BD, $e->getMessage());
    }
}

/**
* Descripción: Inserta una persona en la base de datos
* @param persona Persona para insertar en la base de datos
* @return Indica si se ha insertado la persona correctamente (Código 1)
*/
public function insertarPersona($persona)
{

```

```

try
{
    // Obtengo una instancia de la base de datos ya conectada
    $pdo = ConexionBD::obtenerInstancia()->obtenerBD();
    // Creo la sentencia INSERT
    $comando = "INSERT INTO " . self::NOMBRE_TABLA . " ( " .
        self::DNI . "," .
        self::NOMBRE . "," .
        self::APELLIDOS . "," .
        self::TELEFONO . "," .
        self::EMAIL . ")" .
        " VALUES(?,?,?,?,?)";
    $sentencia = $pdo->prepare($comando);
    // Pongo los datos en la consulta INSERT
    $sentencia->bindParam(1, $persona->DNI);
    $sentencia->bindParam(2, $persona->nombre);
    $sentencia->bindParam(3, $persona->apellidos);
    $sentencia->bindParam(4, $persona->telefono);
    $sentencia->bindParam(5, $persona->email);
    // Ejecuto la consulta
    $resultado = $sentencia->execute();
}
catch (PDOException $e)
{
    throw new ExcepcionApi(self::ESTADO_ERROR_BD, $e->getMessage());
}

switch ($resultado)
{
    case self::ESTADO_CREACION_EXITOSA:
        http_response_code(200);
        return correcto;
        break;
    case self::ESTADO_CREACION_FALLIDA:
        throw new ExcepcionApi(self::ESTADO_CREACION_FALLIDA, "Ha ocurrido un
        error.");
        break;
    default:
        throw new ExcepcionApi(self::ESTADO_FALLA_DESCONOCIDA, "Fallo
        desconocido.", 400);
}
}
}

```

A continuación vemos el fichero **obtenerTodasPersonas**, del directorio raíz del proyecto, el cual será usado al ejecutar la función **obtenerTodasPersonas()** que acabamos de ver en la clase `persona.php` (justo arriba).

```
<?php
// Hago que se muestren los errores si los hay
ini_set('display_errors', 1);
// Importamos la vista y el controlador para poder utilizarlo:
require_once('vistas/VistaJson.php');
require_once('controladores/ControladorPersonas.php');
// Tipo de vista de la salida de datos.
$vista = new VistaJson();
// Con esta función nos aseguramos que cualquier excepción que ocurra se muestre
adecuadamente
// en el mismo formato para evitar problemas.
set_exception_handler(function ($exception) use ($vista)
{
    $cuerpo = array(
        array(
            "estado" => $exception->estado,
            "mensaje" => $exception->getMessage()
        )
    );
    if ($exception->getCode())
    {
        $vista->estado = $exception->getCode();
    }
    else
    {
        $vista->estado = 500;
    }

    $vista->imprimir($cuerpo);
}
);
// Me creo un controlador de personas
$controladorp = new ControladorPersonas();
// Saco por pantalla en formato JSON el resultado, llamando al método
obtenerTodasPersonas:
$vista->imprimir($controladorp->obtenerTodasPersonas());
```

Para llamar al fichero ObtenerTodasPersonas y que ejecute su código, tendremos que ir a nuestro navegador y escribir la ruta a ese fichero:

localhost:50/personas/obtenerTodasPersonas.php

localhost será la raíz de nuestro proyecto, donde se encuentran todos estos ficheros que hemos agregado, de modo que la ruta será:

Donde “50” recordemos que es el puerto que hemos asignado al servidor. Nos mostrará lo siguiente:

```
[
  {
    "estado": 1,
    "mensaje": [
      {
        "DNI": "14786325P",
        "nombre": "Juan",
        "apellidos": "López López",
        "telefono": 666666666,
        "email": "juanlopez@gmail.com"
      },
      {
        "DNI": "24158749L",
        "nombre": "Pepe",
        "apellidos": "Adamuz Núñez",
        "telefono": 666956666,
        "email": "pepeadamuz@gmail.com"
      },
      {
        "DNI": "36214758I",
        "nombre": "Dolores",
        "apellidos": "Pérez Aguilera",
        "telefono": 684766666,
```



```

        "email": "doloresperez@gmail.com"
    }
]
}
]
```

Donde "estado": 1 será creación exitosa, tal como hemos definido en el fichero mensajes.php del directorio datos de nuestro proyecto y dentro del "mensaje" un array con todas las personas de nuestra base de datos.

Ahora veremos cómo obtener una persona con su DNI. En el fichero ControladorPersonas.php tenemos nuestra función obtenerPersonaDNI donde hay un atributo (?) en la sentencia WHERE que deberemos de indicar su valor, y para ello usamos el método bindParam:

```
$sentencia->bindParam(1, $DNI);
```

Indicando la posición de la interrogación y el atributo, ejecutamos la sentencia:

```
$sentencia->execute();
```

Obtenemos todos los resultados y los devolvemos (como vemos en el fichero ControladorPersonas.php...)

Ahora creamos un método obtenerPersonaDNI que es el que tenemos en el fichero que lleva su nombre, en la raíz del proyecto. La función obtenerPersonaDNI al final de este fichero, tiene un parámetro \$dni que se introducirá mediante la url que introdujimos antes, pero con una modificación:

```
localhost:50/personas/obtenerPersonaDNI.php?DNI=14786325P
```

Donde habrá que añadir una **interrogación (?)** después del fichero que contiene el método, seguido del **nombre del parámetro (DNI)** y su **valor (14786325P)**, que será el que tenga en nuestra base de datos personas en la tabla Persona, creada anteriormente.

Obteniendo el siguiente resultado en nuestro navegador (servidor):

```
[
  {
    "estado": 1,
    "mensaje": [
      {
        "DNI": "14786325P",
        "nombre": "Juan",
        "apellidos": "López López",
        "telefono": 666666666,
        "email": "juanlopez@gmail.com"
      }
    ]
  }
]
```

Todo esto se consigue gracias al método:

```
$dni = $_REQUEST['DNI'];
```

Y a la vista, que imprime el resultado obtenido con el controlador (ambos métodos usados en la clase obtenerPersonaDNI que acabamos de ejecutar en el servidor Apache:

```
$vista→imprimir($controladorp→obtenerPersonaDNI($dni));
```

Si queremos insertar una persona, en nuestra clase ControladorPersona tenemos el método insertarPersona(\$persona), en el cual le pasaremos la persona que queramos agregar:

```
// Obtengo una instancia de la base de datos ya conectada
$pdo = ConexionBD::obtenerInstancia()->obtenerBD();
// Creo la sentencia INSERT
$comando = "INSERT INTO " . self::NOMBRE_TABLA . " ( " .
```

```

        self::DNI . "," .
        self::NOMBRE . "," .
        self::APELLIDOS . "," .
        self::TELEFONO . "," .
        self::EMAIL . ")" .
        " VALUES(?,?,?,?,?)";

$sentencia = $pdo->prepare($comando);
// Pongo los datos en la consulta INSERT
$sentencia->bindParam(1, $persona->DNI);
$sentencia->bindParam(2, $persona->nombre);
$sentencia->bindParam(3, $persona->apellidos);
$sentencia->bindParam(4, $persona->telefono);
$sentencia->bindParam(5, $persona->email);
// Ejecuto la consulta
$resultado = $sentencia->execute();

```

Cuando llamemos a este método deberemos de enviar más de un parámetro, en la url del navegador, los cuales se agregarán añadiendo un ampersan (&) seguido del nombre del parámetro igual (=) a su valor. Con el carácter ampersan podremos poner tantos atributos como necesitemos, en nuestro caso deberíamos de poner 5 (DNI, Nombre, Apellidos, Telefono y Email), tal y como aparecen en nuestra clase insertarPersona.php.

Conexión a MYSQL desde PHP7

Como ya sabemos, en este tema usaremos **PHP** como un **intermediario** entre nuestra aplicación **Android** y nuestra **base de datos MySQL** en un servidor «remoto» (aunque estará en local si usamos XAMPP). Vamos a ver cómo podemos acceder con PHP 7 a MySQL.

Como **PHP 7** es un lenguaje de programación orientado a objetos, vamos a crear una **clase ConexionDB** para **conectarlo a la base de datos**.

Por ahora (sabiendo que vamos a utilizar el **Modelo Vista-Controlador** más adelante), crearemos una **carpeta**, llamada «**datos**», que será la encargada de contener todo lo relacionado con el acceso a la base de datos y que contendrá los siguientes ficheros:

- **ConexionBD.php**: Este fichero tendrá una clase que se encargará de la **conexión y desconexión** de la base de datos.
- **login_mysql.php**: Este fichero tendrá todos los **datos** necesarios para realizar la **conexión** a la base de datos.
- **mensajes.php**: Contendrá los **mensajes y variables** que vamos a utilizar en el proyecto.

En el fichero **login_mysql.php**, tendremos que definir:

las variables que nos indicarán el **nombre del host** donde vamos a conectarnos (**localhost**, en nuestro caso),

el nombre de la **base de datos**,

los **datos de conexión del usuario**.

Esto lo podremos hacer con el **método define**, que definirá una **variable global** a nuestro proyecto.

Para realizar la **conexión** a la base de datos, utilizaremos la **clase de PHP 7** que nos provee dicha funcionalidad, a la cual tendremos que **pasarle todos los datos** anteriores. Como estamos utilizando una conexión a una base de datos, tendremos que **desconectarnos** cuando **terminemos** nuestras operaciones, lo cual podremos hacer en el **destructor** de la clase. Un destructor se llamará automáticamente cuando se destruya el objeto. Este **_destructor** estará definido en la clase ConexionBD.php.

Definición de variables en la clase **login_mysql.php**:

```
define("NOMBRE_HOST", "localhost");    // Nombre del host
define("BASE_DE_DATOS", "personas");// Nombre de la base de modelos
define("USUARIO", "root");              // Nombre del usuario
define("CONTRASEÑA", "");               // Contraseña
```

Clase **ConexionBD.php**:

```
<?php

/**
 * Esta clase sirve para poder realizar una conexión y desconexión a la base de datos
 * MySQL de nuestro servidor.
 */

require_once('login_mysql.php');

class ConexionBD
{
    const ESTADO_ERROR_BD = 3;

    /**
     * Atributo para la conexión de la base de datos
     */
    private static $db = null;

    /**
     * Instancia de PDO (extensión Objetos de Datos de PHP,
     * capa de abstracción de acceso a datos)
     */
    private static $pdo;

    /**
     * Constructor de la clase. Conecta con la base de datos.
     */
    final private function __construct()
    {
        try
        {
            // Crear nueva conexión PDO
            self::obtenerBD();
        }
        catch (PDOException $e)
        {
            // Manejo de excepciones
            throw new ExcepcionApi(ESTADO_ERROR_BD, $e->getMessage());
        }
    }
}
```

```

/**
 * Retorna en la única instancia de la clase
 * @return ConexionBD|null
 */
public static function obtenerInstancia()
{
    if (self::$db === null)
    {
        self::$db = new self();
    }
    return self::$db;
}

/**
 * Crear una nueva conexión PDO basada
 * en las constantes de conexión
 * @return PDO Objeto PDO
 */
public function obtenerBD()
{
    if (self::$pdo === null)
    {
        self::$pdo = new PDO(
            'mysql:dbname=' . BASE_DE_DATOS .
            ';host=' . NOMBRE_HOST . ";",
            USUARIO,
            CONTRASENA,
            array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8")
        );

        // Habilitar excepciones
        self::$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }

    return self::$pdo;
}

/**
 * Evita la clonación del objeto
 */
final protected function __clone() {}

/**
 * Destructor de la clase. Cierra la conexión a la base de datos.
 */

```

```

function _destructor()
{
    self::$pdo = null;
}
}

?>

```

Clase **mensajes.php**:

```

<?php

if (!defined('ESTADO_CREACION_EXITOSA'))
    define('ESTADO_CREACION_EXITOSA', 1);
if (!defined('ESTADO_CREACION_FALLIDA'))
    define('ESTADO_CREACION_FALLIDA', 2);
if (!defined('ESTADO_ERROR_BD'))
    define('ESTADO_ERROR_BD', 3);
if (!defined('ESTADO_AUSENCIA_CLAVE_API'))
    define('ESTADO_AUSENCIA_CLAVE_API', 4);
if (!defined('ESTADO_CLAVE_NO_AUTORIZADA'))
    define('ESTADO_CLAVE_NO_AUTORIZADA', 5);
if (!defined('ESTADO_URL_INCORRECTA'))
    define('ESTADO_URL_INCORRECTA', 6);
if (!defined('ESTADO_FALLO_DESCONOCIDO'))
    define('ESTADO_FALLO_DESCONOCIDO', 7);
if (!defined('ESTADO_PARAMETROS_INCORRECTOS'))
    define('ESTADO_PARAMETROS_INCORRECTOS', 8);

if (!defined('error'))
    define('error',
        [
            [
                "estado" => ESTADO_FALLO_DESCONOCIDO,
                "mensaje" => utf8_encode("Error desconocido.")
            ]
        ]
    );

if (!defined('correcto'))
    define('correcto',
        [
            [

```

```
        "estado" => ESTADO_CREACION_EXITOSA,  
        "mensaje" => utf8_encode("OK")  
    ]  
];  
);  
?>
```


Tipo de base de datos remota que conviene utilizar

En cuanto a tipos de bases de datos, no solo existen las bases de datos relacionales, sino que hay otros tipos, como las **bases de datos XML**, las cuales utilizan ficheros XML para almacenar los datos, **prescindiendo de tablas, de claves y de demás conceptos**.

Las **bases de datos relacionales** también pueden utilizarse en servidores remotos, pero tienen el inconveniente de que, **si son muy grandes**, pueden llegar a ser **lentas**, y eso no es muy bueno en una conexión remota.

Para usos de **bases de datos pequeñas o medianas**, podríamos usar una **base de datos relacional**, y, si tuviéramos que utilizar una **base de datos muy grande**, podríamos intentar utilizar otros tipos de bases de datos, ya que, en bases de datos grandes, las relacionales pueden volverse muy lentas. Las bases de datos no relacionales se utilizan en escenarios donde la escalabilidad, la velocidad y la flexibilidad son prioritarias.

Algunas grandes empresas que utilizan bases de datos **no relacionales** son: Facebook, Twitter, Youtube... debido a la ingente cantidad de información que manejan.

Operaciones en MYSQL desde PHP7

Las operaciones que podremos realizar **desde PHP** sobre nuestra **base de datos MySQL** son las siguientes:

- **Conexión** con la base de datos.
- **Desconexión** de la base de datos.
- **Inserción** de datos.
- **Borrado** de datos.
- **Actualización** de datos.
- **Recuperación** de datos.

Mediante el **método execute**:

- podremos ejecutar, mediante sus **instrucciones SQL** pertinentes, las operaciones en datos:
 - **Inserción**,
 - **Borrado**,
 - y **Actualización**.

Las operaciones de **selección de datos** podremos realizarlas mediante una consulta **SELECT** (tan compleja como la necesitemos), y podremos **ejecutarlas mediante el método execute**, que nos devolverá un **cursor** con todos los datos a devolver.

Para este tipo de tareas, utilizaremos el **modelo Vista-Controlador**, como ya comentamos anteriormente, y tendremos una estructura de carpetas como la que mostramos a continuación:

MVC de un proyecto PHP básico:

Explicación Modelo Vista-Controlador de un proyecto PHP básico:

Controladores (Todas las clases que son controladores de nuestros modelos):

ControladorPersonas.php (permitirá realizar operaciones con las personas)

Datos (todo lo que tenga que ver con la conexión a la BBDD):

ConexionBD.php

login_mysql.php

mensajes.php

Modelos (Todas la clases que representen un modelo en nuestro proyecto):

Persona.php

Utilidades (Todas las clases que nos proporcionarán una utilidad diferente):

ExcepcionApi.php (será una API para el control de excepciones en nuestro proyecto)

Vistas (Todos los ficheros que tienen que ver con las vistas):

VistaApi.php

VistaJson.php

(Nos permitirán formatear los resultados con formato JSON)

En la **Carpeta raíz del proyecto** (corresponden con las operaciones que vamos a hacer):

insertarPersona.php

obtenerPersonaDNI.php

obrtenerTodasPersonas.php

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Servicios en red

Codificación JSON

Lenguaje de programación backend

Conexiones HTTPS

Vamos a ver:

- Cómo funcionan las **conexiones HTTPS** y sus principales diferencias con respecto a las conexiones HTTP.
- El **esquema de tipo cliente/servidor** que, aunque ya puede que lo conozcas de otras asignaturas, en esta lo vamos a enfocar hacia las **aplicaciones Android**, ya que necesitan de una **conexión a un servidor externo**.
- Los **permisos** que necesitaremos otorgar a nuestras aplicaciones para que sean capaces de realizar conexiones HTTPS.
- Cómo podemos **obtener datos del tipo JSON** mediante una conexión HTTPS a un servidor externo, para poder **procesar en nuestras aplicaciones** Android.

Conexiones HTTP Y HTTPS

Cuando hablamos de conexiones HTTP, lo primero que posiblemente nos venga a la mente sea una visita a una página web a través de Internet, mediante un ordenador, pero, aparte de los ordenadores, hay numerosos dispositivos que también pueden realizar este tipo de conexiones, y como podremos adivinar, los smartphones son uno de ellos.

Hoy en día, el uso de un smartphone, sea del tipo que sea, es algo rutinario. Estos dispositivos tienen conexión a Internet, pudiendo utilizar aplicaciones muy comunes, como YouTube, WhatsApp, Telegram, Chrome e infinidad de juegos que nos podemos descargar y que hacen uso de similares tecnologías.

Cuando hablamos de **HTTP**, nos estamos refiriendo al protocolo que lleva su mismo nombre y que nos va a permitir realizar conexiones remotas entre dispositivos. Este protocolo quedó obsoleto hace mucho tiempo, aunque aún se siga utilizando, ya que **no es seguro**, es decir, **no cifra la información** antes de enviarla, lo que puede provocar que **alguien pueda acceder** a lo que se está enviando o recibiendo.

Esto se solucionó con la aparición del protocolo **HTTPS (Hypertext Transfer Protocol Secure)**, que sí utiliza un **cifrado SSL/TLS**, mucho más apropiado para el tráfico de información en este tipo de canales. Con este tipo de cifrado, se consigue que **ciertas informaciones**, como las contraseñas, **no puedan ser captadas por un tercero** de una forma tan sencilla como con HTTP.

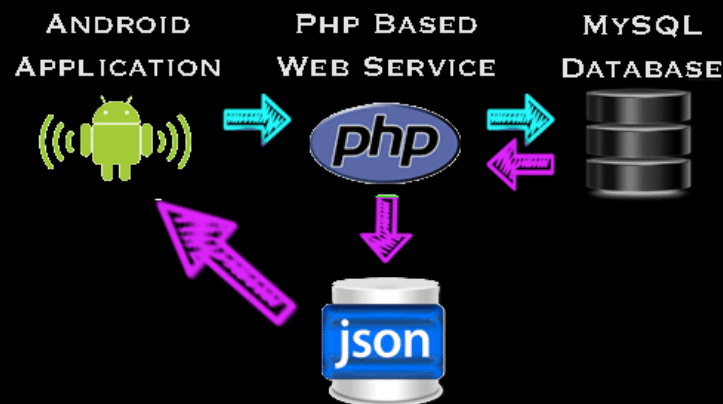
El **puerto** que utiliza el protocolo **HTTPS** es el **443** (HTTP usa el 80).

Como ya hemos comentado, absolutamente todos los smartphones actuales disponen de una **tarjeta de red integrada** que hará uso de este protocolo (entre otros) y que les permitirán realizar conexiones a Internet a unas velocidades muy altas.

Esquema cliente/servidor en aplicaciones Android

Todas las aplicaciones Android que vamos a realizar en esta unidad van a seguir el esquema que se muestra en la siguiente figura:

Esquema Android - PHP - JSON - MySQL



Este esquema consta de las siguientes **partes**:

1. **Aplicación Android**: Esta será nuestra aplicación instalada en nuestro smartphone.
2. **Servicio web basado en PHP**: Esta parte la componen todos los ficheros PHP que van a formar la **API REST**, y será con la que **accederemos a la parte de la base de datos**. Una API REST proporciona una forma estandarizada y eficiente para que **diferentes sistemas se comuniquen** a través de la web mediante el uso de los principios de REST y el protocolo HTTP.
3. **Base de datos MySQL**: Esta será la base de datos MySQL en la que estarán todas las tablas y datos en un **servidor remoto**.
4. **JSON**: Este será el **lenguaje intermedio** que permitirá la comunicación de la aplicación con la API REST en PHP.

El funcionamiento será el siguiente:

1. Cuando la **aplicación** necesite cierta información, **llamará a la API REST** en **PHP** mediante una **conexión HTTPS** a través de una **URL**.
2. La **API REST** realizará la operación solicitada por la aplicación **accediendo** a la **base de datos MySQL**.
3. Una vez que la API REST tenga la respuesta solicitada la **codificará en JSON** y la **devolverá** a nuestra aplicación Android.
4. La **aplicación Android** obtendrá dicha información codificada en **JSON**, la **decodificará** y podrá **tratar la información** obtenida.

Diferentes formatos, mismo esquema

*Ya comentamos en unidades pasadas que la información suele codificarse en JSON o en XML. En caso de que **necesitemos una codificación XML** podremos utilizar exactamente el **mismo esquema** cliente-servidor **que con JSON**, con la única variante de que la información **se devolverá en formato XML**.*

Transformando datos

Siempre que vayamos a usar **conexiones vía HTTPS** vamos a **comunicar una aplicación con el servidor utilizando JSON**, pero el formato JSON no es válido para trabajar en Android.

La actividad consiste por tanto en: dado un **fichero** con una serie de **datos de personas** codificado en **JSON**, **convertirlo a un ArrayList** y **mostrarlo en una lista optimizada**. Cada persona tiene un DNI, un nombre y unos apellidos.

En primer lugar, sí que debemos aprender a realizar este proceso, ya que lo vamos a necesitar realizar cada vez que nuestra aplicación realice una petición a un servidor, además, no es un proceso muy complicado, pero sí vital.

Este **proceso** se puede dividir en varias partes.

1. En primer lugar, deberemos crear una **clase** que represente la **información que vamos a obtener** del servidor, una clase Persona en este caso.
2. Una vez hecho, deberemos **recorrer el JSON**, que será un array, ya que podremos obtener varias personas, y **elemento a elemento** del array, en este caso, persona a persona, podremos ir **obteniendo todas sus propiedades** para **crear un objeto Persona** completo.
3. Por último, una vez tengamos la **persona creada**, deberemos **agregarla a un array**.

Puedes ver un ejemplo en pseudocódigo:

Transformando un array JSON en un array

```
// Convertimos el array json
desde i = 0 hasta json.tamaño
    dni = arrayjson.getElemento(i).obtenerDNI()
    nombre = arrayjson.getElemento(i).obtenerNombre()
    apellidos = arrayjson.getElemento(i).obtenerApellidos()
    persona = Persona(dni, nombre, apellidos)
    arrayPersona.agregar(persona)
// Mostramos el arrayjson.getElemento
desde i = 0 hasta arrayPersona.tamaño
    persona = arrayPersona.getElemento(i)
    mostrar(persona)
```

Permisos para conectar a Internet

El Sistema Operativo Android trabaja de una forma peculiar con respecto a sus aplicaciones, ya que estas se ejecutan de una forma distinta de las que se ejecutan, por ejemplo, en Windows. En este último sistema operativo, cualquier aplicación puede utilizar todos los recursos del ordenador con total libertad, cosa que no ocurre en los smartphones Android, por seguridad y debido a que los recursos son muy limitados.

En el desarrollo de aplicaciones Android, vamos a encontrarnos con que este sistema operativo usa un régimen de **permisos**, consistentes en **restricciones** que se adjudican a todas las aplicaciones, y que provocarán que estas no utilicen de forma indebida ciertos recursos como, por ejemplo, la cámara, el almacenamiento (tanto interno como externo), el bluetooth, el micrófono y un largo etcétera.

Cualquier aplicación Android necesitará que se le concedan permisos para poder utilizar los recursos de forma predeterminada. Esto se hará en el **momento de la instalación** de la misma, en el que nos aparecerá una lista de los permisos que dicha aplicación va a utilizar y que podremos examinar cuidadosamente antes de aceptar. Una vez **aceptados**, la aplicación **podrá utilizar** los recursos que indican sus permisos.

En nuestro caso, a la hora de desarrollar una aplicación, deberemos indicar los permisos en el manifiesto de la misma, fichero manifest.xml.

En el siguiente código xml podemos ver los permisos necesarios para que nuestra aplicación pueda tener una conexión a Internet.

Permisos de conexión a Internet

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.ejemplovideo1">

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

...

```

Obtener JSON desde Android vía HTTP

Los conceptos teóricos estudiados hasta ahora ya nos permiten crear una aplicación Android que se **conecte a un servidor remoto**, donde exista una **API REST creada con PHP 7**, que accederá a una **base de datos MySQL** y codifica en **JSON**.

Para ello, vamos a **crear las siguientes clases**:

- **JSONParser**: Esta clase nos permitirá **obtener los datos con el formato JSON** de una **URL** específica, pudiendo pasarle parámetros por medio del **método POST**. Esta clase siempre se implementará de igual forma.
- **ServidorPHPException**: Debido a que el uso de **conexiones HTTPS** pueden provocar una gran lista de excepciones, vamos a crear una excepción que lanzaremos cuando alguna de estas ocurra, teniendo la gestión de errores centralizada en una sola excepción y no en muchas, que puede resultar muy tedioso.

Siempre que necesitemos utilizar **conexiones de tipo HTTPS**, necesitaremos usar el sistema de hilos, es decir, toda conexión del tipo HTTPS deberá ser **ejecutada mediante un hilo** independiente del resto de la aplicación.

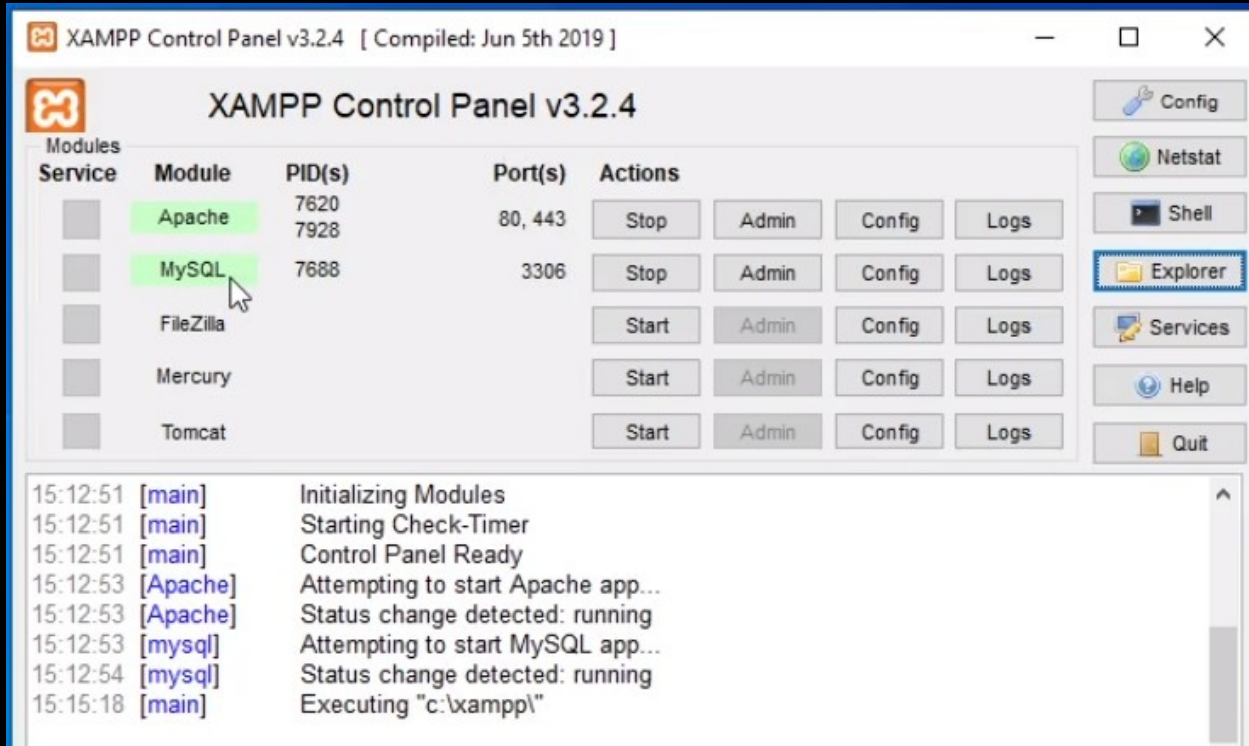
Con la **clase JSONParser**, vamos a forzar a nuestra aplicación a **esperar** a que la **petición HTTPS** se realice para poder continuar. Esto **no es lo más adecuado**, pero para no sofisticar mucho las primeras aplicaciones que vamos a desarrollar, lo implementaremos así.

Dentro de la clase JSONParser, vamos a crear los siguientes **métodos**:

- **getJSONArrayFromUrl**: A este método le pasaremos la **URL** de la página que ejecuta el **servicio** en el servidor y un **array** con los **parámetros POST** necesarios. Nos devolverá un **objeto JSONArray** con todos los datos **devueltos** por el servicio en **formato JSON**.
- **getJSONObjectFromUrl**: De igual modo que el método anterior, a este le pasaremos la **URL** de la página que ejecuta el servicio en el **servidor** y un array con los **parámetros POST** necesarios. No obstante, en este caso, nos devolverá un **objeto JSONObject** con todos los datos **devueltos** por el servicio en **formato JSON**.
- **buildURL**: Este método privado **construye una URL** con la información que le facilitemos. Le pasaremos como parámetros la **URL base** y un **mapa** con todos los **parámetros** a pasar, **devolviéndonos la URL formateada** correctamente.

Cómo obtener datos JSON mediante una consulta HTTP mediante la biblioteca Volley:

Tendremos un servidor externo en una máquina virtual con XAMPP instalado:

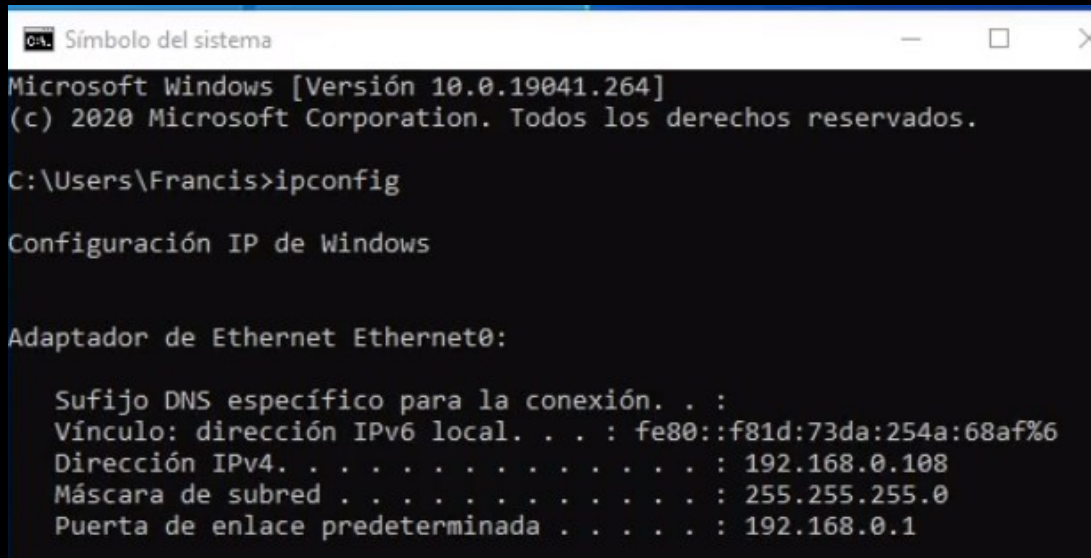


Con esta estructura de ficheros en la parte del servidor:

```
Símbolo del sistema
(c) 2020 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Francis>tree C:\xampp\htdocs\personas /f
Listado de rutas de carpetas
El número de serie del volumen es 88D3-22A4
C:\XAMPP\HTDOCS\PERSONAS
    insertarPersona.php
    obtenerPersonaDNI.php
    obtenerTodasPersonas.php
    --controladores
        ControladorPersonas.php
    --datos
        ConexionBD.php
        login_mysql.php
        mensajes.php
    --modelos
        Persona.php
    --utilidades
        ExcepcionApi.php
    --vistas
        VistaApi.php
        VistaJson.php
```

Lo primero que debemos de configurar es la IP de nuestra máquina, ya que al ser un servidor debe de tener una IP estática. Mediante el comando ipconfig en Windows podremos ver nuestra IP, que en este caso es la Dirección IPV4 que vemos en la captura:



```
Microsoft Windows [Versión 10.0.19041.264]
(c) 2020 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Francis>ipconfig

Configuración IP de Windows

Adaptador de Ethernet Ethernet0:

    Sufijo DNS específico para la conexión. . . : 
    Vínculo: dirección IPv6 local. . . : fe80::f81d:73da:254a:68af%6
    Dirección IPv4. . . . . : 192.168.0.108
    Máscara de subred . . . . . : 255.255.255.0
    Puerta de enlace predeterminada . . . . . : 192.168.0.1
```

En la base de datos, que consultamos desde phpMyAdmin, vemos que tenemos la base de datos “personas” con una tabla llamada persona con los datos que se ven en la imagen:



Desde el fichero build.gradle (Module:app) colocaremos la biblioteca volley como se muestra a continuación:

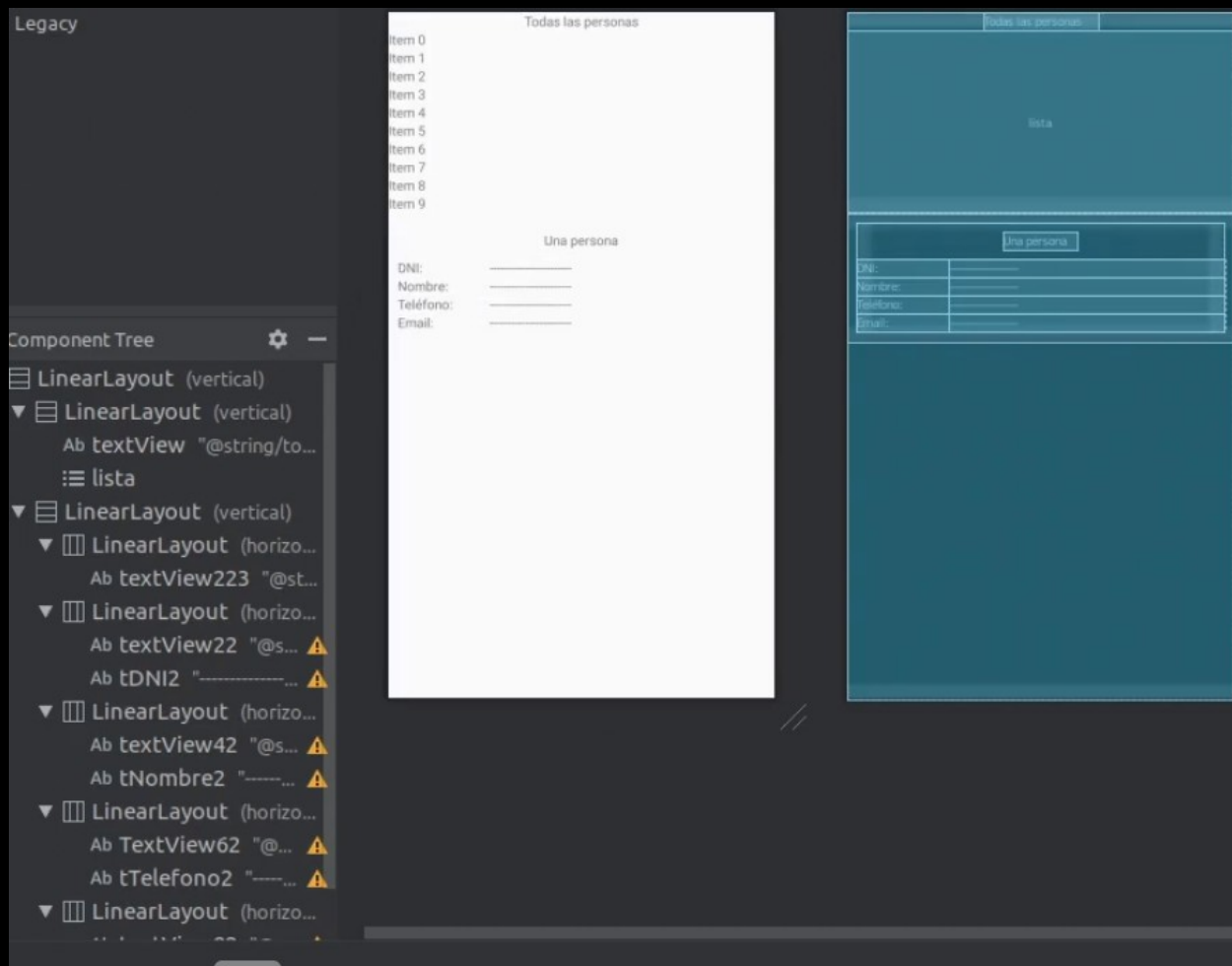
```
// Biblioteca Volley para la obtención de JSON desde una URL
implementation 'com.android.volley:volley:1.1.1'
```

Después tendremos que reconfigurar el proyecto para que se pueda usar esta biblioteca.
Lo siguiente que debemos hacer es conceder permisos de acceso a internet:

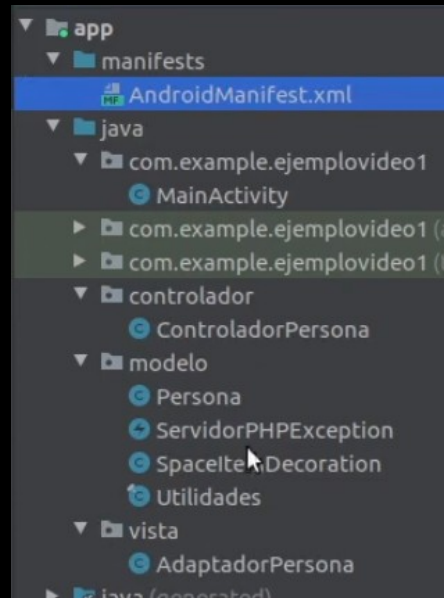
```
<uses-permission android:name="android.permission.INTERNET"/>  
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

Este fichero es el AndroidManifest.xml, dentro de la carpeta manifest.

La aplicación tiene una lista donde mostraremos a todas las personas y una serie de etiquetas con las que mostraremos los datos de cada persona, obtenida por su DNI:



Estamos usando el modelo vista controlador:



Dentro de “modelo” tenemos una clase llamada utilidades, en la que tenemos lo siguiente:

- Una serie de variables con los posibles resultados,
- Una variable con la URL de nuestro servidor, es decir, una url con la IP de nuestro servidor,
- Un método buildURL, que nos formateará una url en un formato válido.

```
public final class Utilidades
{
    // Variables con los posibles resultados
    public static final int RESULTADO_OK = 1;
    public static final int RESULTADO_ERROR = 2;
    public static final int RESULTADO_ERROR_DESCONOCIDO = 3;
    // Variable con la url de nuestro servidor, con la IP de nuestro servidor y la carpeta
    // personas con toda la información
    public static final String URLSERVIDOR = "http://192.168.0.107/personas/";

    /**
     * Crea una URL válida con parámetros
     * @param url URL base
     * @param params Parámetros para la URL
     * @return URL formateada con sus parámetros
     */
    public static String buildURL(String url, HashMap<String, String> params)
    {
```

```

Uri.Builder builder = Uri.parse(url).buildUpon();
if (params != null)
{
    for (Map.Entry<String, String> entry : params.entrySet())
    {
        builder.appendQueryParameter(entry.getKey(), entry.getValue());
    }
}
return builder.build().toString();
}
}

```

Además hemos creado también en “modelo” una clase Persona que será la que moldeará nuestro dato-objeto persona que tenemos en la base de datos:

```

public class Persona
{
    private String DNI, nombre, apellidos, telefono, email;

    public Persona(String DNI, String nombre, String apellidos, String telefono, String email) {
        this.DNI = DNI;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.telefono = telefono;
        this.email = email;
    }

    public String getDNI() {
        return DNI;
    }

    public void setDNI(String DNI) {
        this.DNI = DNI;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}

```



```

public String getApellidos() {
    return apellidos;
}

public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
}

public String getTelefono() {
    return telefono;
}

public void setTelefono(String telefono) {
    this.telefono = telefono;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}
}

```

Y tenemos también dentro del paquete “controlador” una clase ControladorPersona que será la encargada de gestionar las operaciones con personas, que tendrá lo siguiente:

- dos atributos de tipo String que guardarán la url del fichero .php que queramos llamar,
- Una variable de tipo Context, que será el contexto de la aplicación,
- Un RecyclerView,
- Un objeto de tipo AdaptadorPersona.

```

public class ControladorPersona {
    // atributos que guardarán la url del fichero que queramos llamar
    private final String URLOBTENERPERSONAS = URLSERVIDOR + "obtenerTodasPersonas.php";
    private final String URLOBTENERPERSONA = URLSERVIDOR + "obtenerPersonaDNI.php";

    private Context contexto;
    private RecyclerView lista;
    private AdaptadorPersona adaptador;
}

```

A nuestra clase le pasaremos en el constructor un objeto de tipo Context y la lista donde queremos mostrar las personas, ya que todo esto se hará mediante un hilo diferente al hilo principal de la aplicación, y deberemos de pasar los atributos gráficos donde queremos que se muestre la información, en este caso la lista y luego también las etiquetas:

```
public ControladorPersona(Context contexto, RecyclerView lista)
{
    this.contexto = contexto;
    this.lista = lista;
}
```

También hemos creado el método obtenerTodasPersonas, que utilizando la biblioteca Volley obtendrá mediante http los datos en nuestra url. Para ello deberemos crear una cola de peticiones de Volley e inicializarla con nuestro contexto y mediante un objeto de tipo JSONArrayRequest podremos obtener la petición que queramos. Esto lo haremos a partir del try donde podemos obtener el resultado de nuestra url y saber si su estado es correcto. Si el estado es correcto obtendremos todos los datos en un objeto JSONArray e iremos obteniendo JSONObject por JSONObject y mediante los métodos get de su tipo de dato correspondiente obtendremos los datos de cada persona, agregándola a un array.

Una vez terminado esto, cerraremos la lista de una forma normal y mostramos el resultado mediante el método refrescar. Es importante que se haga en este punto ya que de lo contrario no podremos hacerlo al ser un hilo diferente:

```
/**
 * Obtiene todas las personas del servidor
 * @throws ServidorPHPException Esta excepción se lanzará si ocurre algún error en la conexión
 */
public void obtenerTodasPersonas() throws ServidorPHPException
{
    try
    {
        // Inicializo la cola de peticiones, utilizamos la biblioteca volley para obtener los datos en
        // nuestra url, para ello creamos la cola de volley
        RequestQueue colavolley = Volley.newRequestQueue(contexto);
        String url = URLOBTENERPERSONAS;
        // Mediante un objeto JSONArrayRequest podemos obtener la petición que queramos
        JSONArrayRequest jsonObjectRequest = new JSONArrayRequest(
            Request.Method.GET,
            url,
            null,
            new Response.Listener<JSONArray>()
            {
                @Override
                public void onResponse(JSONArray response)
            }
        );
    }
}
```

```

{
    // Procesamos el JSONArray
    try
    {
        if( response != null )
        {
            int resultadoobtenido = response.getJSONObject(0).getInt("estado");
            //System.out.println("EL RESULTADO ES " + resultadoobtenido);

            switch(resultadoobtenido)
            {
                case RESULTADO_OK:
                    ArrayList<Persona> personas = new ArrayList<>();
                    JSONArray datospersonas =
                        response.getJSONObject(0).getJSONArray("mensaje");
                    for (int i = 0; i < datospersonas.length(); i++)
                    {
                        JSONObject per = datospersonas.getJSONObject(i);
                        String DNI = per.getString("DNI");
                        String nombre = per.getString("nombre");
                        String apellidos = per.getString("apellidos");
                        String telefono = per.getString("telefono");
                        String email = per.getString("email");
                        Persona persona =
                            new Persona(DNI, nombre, apellidos, telefono, email);
                        personas.add(persona);
                    }

                    // Hay que crear en la caperta values un fichero dims.xml y crear ahí
                    // list_space
                    lista.addItemDecoration(new SpacelItemDecoration(
                        contexto, R.dimen.list_space, true, true));
                    // Con esto el tamaño del recyclerview no cambiará
                    lista.setHasFixedSize(true);
                    // Creo un layoutManager para el recyclerview
                    LinearLayoutManager llm = new LinearLayoutManager(contexto);
                    lista.setLayoutManager(llm);

                    adaptador = new AdaptadorPersona(contexto, personas);
                    lista.setAdapter(adaptador);
                    // mostraremos aquí el resultado mediante el método refrescar
                    // Si no lo hacemos aquí no se podrá, al ser un hilo diferente
                    adaptador.refrescar();
                }
            }
        }
    }
}

```

```

        break;
    case RESULTADO_ERROR:
        throw new ServidorPHPException("Error, datos incorrectos.");
    case RESULTADO_ERROR_DESCONOCIDO:
        throw new ServidorPHPException("Error obteniendo los datos del
            servidor.");
    }
}
else
{
    throw new ServidorPHPException("Error obteniendo los datos del servidor.");
}
}
catch (JSONException | ServidorPHPException error)
{
    System.out.println("Error -> " + error.toString());
}
}
},
new Response.ErrorListener()
{
    @Override
    public void onErrorResponse(VolleyError error)
    {
        System.out.println("Error -> " + error.toString());
    }
}
);

```

Una vez terminemos agregamos la petición que hemos creado a la cola de la biblioteca Volley y se ejecutara en un hilo diferente:

```

// Agregamos la petición a la cola para que se ejecute, en un hilo diferente
colavolley.add(jsonObjectRequest);
}
catch (Exception error)
{
    throw new ServidorPHPException(error.toString());
}
}

/**
 * Obtiene una persona del servidor según su DNI
 * Al ejecutarse en un hilo aparte, debemos pasar DNI y todos los TextView donde se mostrarán los

```

```

* datos
* @param DNI DNI de la persona
* @param tDNI Etiqueta donde se mostrará el DNI
* @param tNombre Etiqueta donde se mostrará el nombre
* @param tTelefono Etiqueta donde se mostrará el teléfono
* @param tEmail Etiqueta donde se mostrara el email
* @throws ServidorPHPException Esta excepción se lanzará si ocurre algún error en la conexión
*/

```

El método obtenerPersonaDNI obtendrá los datos de una persona por su DNI y su funcionamiento es casi igual al método obtenerTodasPersonas.

Podemos ver que hemos pasado el DNI de la persona a buscar y todos los TextViews donde se mostrarán los datos, ya que al ejecutarse en un hilo diferente deberemos hacerlo así, igual que el el anterior método mostrábamos la lista.

En primer lugar volveremos a crear un cola de Volley para las peticiones y crearemos nuestro HashMap con los datos pertinentes, en este caso el DNI que será el DNI de la persona que queramos obtener. Construiremos el objeto url con el método buildURL de la clase Utilidades y una vez tengamos esto volvemos a crear un objeto del tipo JsonRequest de la misma forma.

```

public void obtenerPersonaDNI(String DNI, final TextView tDNI, final TextView tNombre, final
    TextView tTelefono, final TextView tEmail) throws ServidorPHPException
{
    try
    {
        // Inicializo la cola de peticiones
        RequestQueue colavolley = Volley.newRequestQueue(contexto);
        // Declaro el array HashMap de los parámetros con el DNI que qeremos obtener
        HashMap<String, String> parametros = new HashMap<>();
        // Meto los parámetros
        parametros.put("DNI", DNI);
        String urlfinal = Utilidades.buildURL(URLOBTENERPERSONA, parametros);
        JsonRequest jsonObjectRequest = new JsonRequest(
            Request.Method.GET,
            urlfinal,
            null,
            new Response.Listener<JSONArray>()
            {
                @Override
                public void onResponse(JSONArray response)
                {
                    try
                    {
                        if( response != null )
                        {

```

```

// Esta parte es muy importante: con estado y mensaje comprobamos si nuestras
// petición es correcta o no:
int resultadoobtenido = response.getJSONObject(0).getInt("estado");
//System.out.println("EL RESULTADO ES " + resultadoobtenido);
switch(resultadoobtenido)

```

Una vez se haya ejecutado, miramos si el estado es correcto, y de ser así obtenemos los datos de la persona y los mostramos en sus textos.

```

{
    case RESULTADO_OK:
        JSONArray datospersona =
            response.getJSONObject(0).getJSONArray("mensaje");
        JSONObject persona = datospersona.getJSONObject(0);
        tDNI.setText(persona.getString("DNI"));
        tNombre.setText(persona.getString("nombre") + " " +
            persona.getString("apellidos"));
        tTelefono.setText(persona.getString("telefono"));
        tEmail.setText(persona.getString("email"));
        break;
    case RESULTADO_ERROR:
        throw new ServidorPHPException("Error, datos incorrectos.");
    case RESULTADO_ERROR_DESCONOCIDO:
        throw new ServidorPHPException("Error obteniendo los datos del" +
            "servidor.");
}
}
else
{
    throw new ServidorPHPException("Error obteniendo los datos del servidor.");
}
}
catch (JSONException | ServidorPHPException e)
{
    System.out.println("Error -> " + e.toString());
}
}
},
new Response.ErrorListener()
{
    @Override
    public void onErrorResponse(VolleyError error)
    {
        System.out.println("Error -> " + error.toString());
    }
}

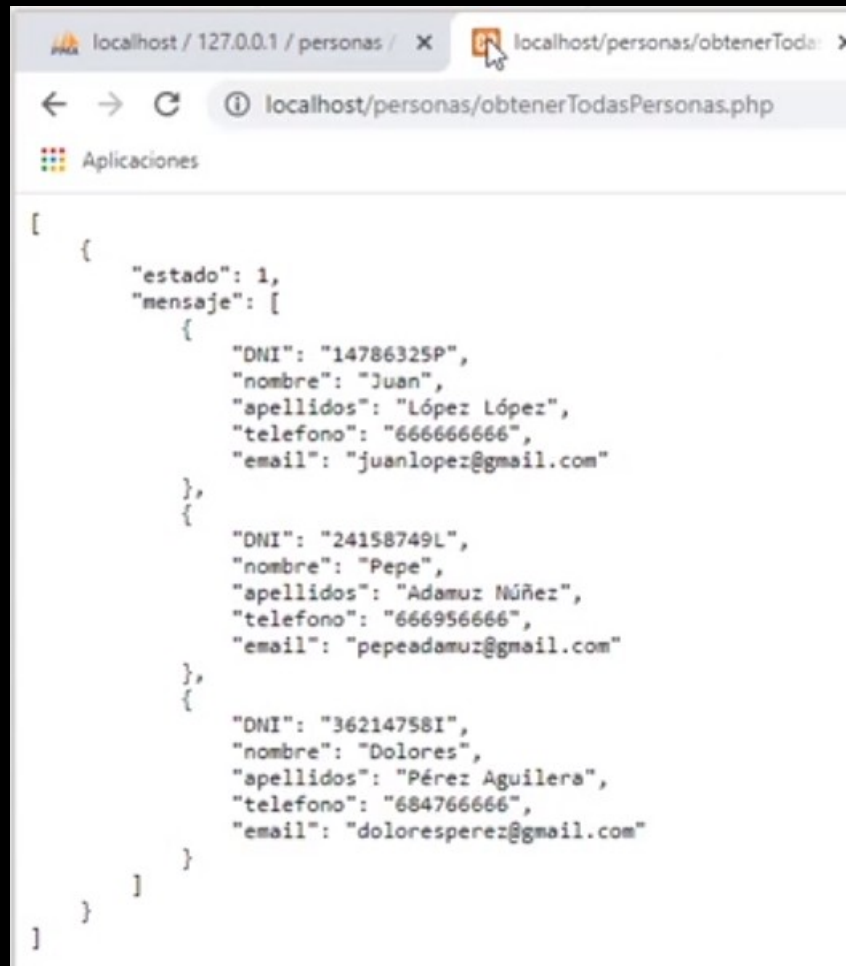
```

```
    }  
    );  
    // Agregamos la petición a la cola para que se ejecute
```

Cuando terminemos de realizar lo que queramos en el código, deberemos de agregar esa petición a la cola Volley. Esta parte es muy importante ya que mediante el estado y el mensaje sabremos si nuestra petición es correcta o no.

```
        colavolley.add(jsonObjectRequest);  
    }  
    catch(Exception e)  
    {  
        throw new ServidorPHPException("Error -> " + e.toString());  
    }  
}  
}
```

Si desde nuestro navegador, teniendo el servidor Apache corriendo con XAMPP, llamamos al fichero `obtenerTodasPersonas.php`, veremos en las respuestas siempre: en una parte el “estado” que nos dirá si se ejecutó bien, y en otra parte que es el “mensaje”, que si se ejecuta correctamente, estará el resultado de la petición, y si hay algún error, en el mensaje estará el error ocurrido.



Hecho todo esto solo nos queda crearnos en el MainActivity.java un controlador de Persona y llamar a sus dos métodos obtenerTodasPersonas y obtenerPersonaDNI:

```
public class MainActivity extends AppCompatActivity {

    private RecyclerView lista;
    private TextView tDNI2, tNombre2, tTelefono2, tEmail2;

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

        // **** Ligamos los recursos de la aplicación ****
        lista = findViewById(R.id.lista);
        tDNI2 = findViewById(R.id.tDNI2);
        tNombre2 = findViewById(R.id.tNombre2);
    }
}
```



```

tTelefono2 = findViewById(R.id.tTelefono2);
tEmail2 = findViewById(R.id.tEmail2);
// *****

ControladorPersona controladorp = new ControladorPersona(this, lista);

try
{
    controladorp.obtenerTodasPersonas();
    controladorp.obtenerPersonaDNI("24158749L", tDNI2, tNombre2, tTelefono2,
tEmail2);
}
catch (ServidorPHPException e)
{
    e.printStackTrace();
}

}
}

```

Al ejecutar la aplicación vemos los datos de todas las personas y los datos de una única persona con DNI 24158749L:

Ejemplo Volley	
Todas las personas	
DNI:	14786325P
Nombre:	Juan López López
Teléfono:	666666666
Email:	juanlopez@gmail.com
DNI:	24158749L
Nombre:	Pepe Adamuz Núñez
Teléfono:	666956666
Email:	pepeadamuz@gmail.com
DNI:	36214758I
Nombre:	Dolores Pérez Aguilera
Teléfono:	684766666
Email:	doloresperez@gmail.com
Una persona	
DNI:	24158749L
Nombre:	Pepe Adamuz Núñez
Teléfono:	666956666
Email:	pepeadamuz@gmail.com

Paso de parámetros en la petición

Ya hemos visto cómo podemos obtener los datos de un servidor remoto mediante una conexión HTTPS, pero ¿y si necesitamos **pasar ciertos parámetros a la petición HTTPS**?

Pongámonos en situación: necesitamos implementar un registro de usuarios, que se guardarán en la base de datos del servidor remoto. En este caso, nos vamos a ver obligados a **pedir la información de los datos al usuario** que se registra y, de alguna forma, deberemos **pasarla como parámetros** en la **petición HTTPS** que hagamos a nuestro servidor.

Esto es posible gracias a los **parámetros POST de PHP**. Estos ya los vimos en unidades pasadas y nos van a ofrecer la posibilidad de **pasar parámetros en las URL**, que podremos **rescatar en el código PHP** mediante la instrucción **`$_REQUEST`**.

Estos parámetros los podremos **pasar a los métodos `getJSONObjectFromURL` y `getJSONArrayFromURL`** como un **objeto de tipo `HashMap<String, String>`** en el que **guardaremos los datos** que queramos pasar a nuestra API REST.

Los **HashMap** nos van a permitir almacenar datos del tipo **clave-valor**, es decir, cada elemento del HashMap tendrá dos valores: una **clave que será única** y un **valor asociado** a dicha clave.

Esto lo podremos utilizar de forma que la **clave** sea el **nombre del parámetro** que queremos pasar y el **valor** sea el propio **valor del elemento**.

Podremos usar el **método `buildURL`**, al que le pasaremos la **dirección base**, es decir, sin parámetros, y un **objeto HashMap** con todos los **parámetros** que necesitemos. Este creará y formateará de forma automática la **dirección final**, devolviéndola lista para usar.

Importante: Hay que tener en cuenta que el **nombre del parámetro POST** debe tener exactamente el mismo nombre que el **valor que obtenemos con `$_REQUEST` en PHP**.

La importancia de la parte backend

Cuando se trata de desarrollar una **aplicación con parte backend**, la complejidad de las mismas se multiplica, ya que hay que realizar tanto la parte de servidor como la parte de la aplicación en sí y, además, cuidar que **ambas se comuniquen correctamente**.

Si realizamos de forma correcta la parte de backend (la del servidor) siguiendo las **pautas del Modelo-Vista-Controlador** y **mostrando** la información de forma correcta en **JSON**, con **códigos de error** que puedan indicar posibles errores en las conexiones, tendremos lo que se conoce como una **API REST totalmente funcional** que podremos aprovechar para que pueda ser **utilizada por otros dispositivos** como, por ejemplo, smartphones de Apple, aplicaciones de escritorio o incluso otros servicios web.

De hecho, la parte de **backend** suele desarrollarse por un **equipo diferente al que desarrolla las aplicaciones móviles** y es lo único que tienen que compartir con las aplicaciones. Es la forma de **comunicarse**, es decir, la parte de **mostrar los datos en formato JSON**.

Ejemplo completo de aplicación Android conectando a servidor remoto mediante conexiones HTTPS (con JSON y MYSQL)

El **servidor remoto** lo vamos a **simular** mediante un servidor local **XAMPP**, teniendo, además, las siguientes consideraciones:

- Al ser un servidor, deberemos **configurar la IP** de nuestra máquina como **fija**, para evitar estar cambiándola cada vez que reiniciemos.
- Tanto el ordenador que haga de **servidor** local y el **smartphone** con el que hagamos la prueba, ya sea un emulador o un dispositivo real, deberán estar **conectados a la misma red**. En **caso contrario**, **no serán visibles** entre ellos.

La funcionalidad que vamos a programar es la de **obtener todos los datos** de todas las personas que haya en el servidor, y **mostrarlas en una lista** dentro de nuestra aplicación.

También vamos a obtener los datos de **una única persona**, seleccionándola **mediante su DNI** y **mostrando** dichos datos en pantalla.

Para ello, vamos a utilizar el Modelo-Vista-Controlador tanto en la parte **backend** como en el **frontend**: nuestra aplicación Android.

La **base de datos** que vamos a utilizar será muy simple y solo tendrá **una tabla** donde se almacenarán los datos de las **personas** que obtendremos más adelante.

Continuando con la estructura del ejemplo articulado en el vídeo anterior, en el siguiente podrás ver el ejemplo completo de la aplicación Android conectándose al servidor. En el apartado de «Recursos del tema» podrás acceder a todo el código generado.

Este ejemplo es igual que el anterior, pero con un fichero en el paquete “modelo” llamado JSONParser.java:

Esta clase nos va a permitir, mediante los métodos `getJSONArrayFromUrl` y `getJSONObjectFromUrl` obtener datos en formato JSON de una url:

```
/**
 * Esta clase implementa la traducción JSON a un servidor web
 */
public class JSONParser
{
    public static final String TAG = "JSONParser";

    /**
     * Constructor de la clase
     */
    public JSONParser() {}
}
```

```

/**
 * Conecta con el servidor y devuelve un JSONArray con los datos obtenidos
 * @param direccionurl URL del servidor
 * @param parametros Parámetros de la consulta
 * @return JSONArray con los resultados de la consulta al servidor
 */
public JSONArray getJSONArrayFromUrl(String direccionurl, HashMap<String, String>
parametros) throws JSONException, IOException
{
    URL url;
    StringBuilder response = new StringBuilder();
    url = new URL(buildURL(direccionurl, parametros));

    HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
    urlConnection.setReadTimeout(15000);
    urlConnection.setConnectTimeout(15000);
    urlConnection.setRequestMethod("GET");
    urlConnection.setRequestProperty("Content-type", "application/json");

    int responseCode = urlConnection.getResponseCode();

    if(responseCode == HttpURLConnection.HTTP_OK)
    {
        InputStream in = new BufferedInputStream(urlConnection.getInputStream());
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));

        String line;
        while ((line = reader.readLine()) != null)
        {
            response.append(line);
        }

    }
    urlConnection.disconnect();

    return new JSONArray(response.toString());
}

/**
 * Conecta con el servidor y devuelve un JSONObject con los datos obtenidos
 * @param direccionurl URL del servidor
 * @param parametros Parámetros de la consulta
 * @return JSONArray con los resultados de la consulta al servidor
 */

```

```

    public JSONObject getJSONObjectFromUrl(String direccionurl, HashMap<String, String>
parametros) throws JSONException, IOException
    {
        URL url;
        StringBuilder response = new StringBuilder();
        url = new URL(buildURL(direccionurl, parametros));

        HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
        urlConnection.setReadTimeout(15000);
        urlConnection.setConnectTimeout(15000);
        urlConnection.setRequestMethod("GET");
        urlConnection.setRequestProperty("Content-type", "application/json");

        int responseCode = urlConnection.getResponseCode();

        if(responseCode == HttpURLConnection.HTTP_OK)
        {
            InputStream in = new BufferedInputStream(urlConnection.getInputStream());
            BufferedReader reader = new BufferedReader(new InputStreamReader(in));

            String line;
            while ((line = reader.readLine()) != null)
            {
                response.append(line);
            }

        }
        urlConnection.disconnect();

        return new JSONObject(response.toString());
    }

    /**
     * Crea una URL válida con parámetros
     * @param url URL base
     * @param params Parámetros para la URL
     * @return URL formateada con sus parámetros
     */
    private String buildURL(String url, HashMap<String, String> params)
    {
        Uri.Builder builder = Uri.parse(url).buildUpon();
        if (params != null)
        {
            for (Map.Entry<String, String> entry : params.entrySet())
            {

```

```
        builder.appendQueryParameter(entry.getKey(), entry.getValue());
    }
}
return builder.build().toString();
}
}
```

Comprobación de conexión a internet en app android

Podemos comprobar si estamos utilizando datos móviles en nuestra conexión a Internet, y en caso afirmativo preguntar al usuario si quiere seguir navegando aunque se esté usando la tarifa de datos.

Un ejemplo para comprobar si estamos usando conexión de datos para, después, preguntar al usuario lo tenemos en el siguiente ejemplo de código en, por ejemplo, nuestro MainActivity.java:

Ejemplo de comprobación de conexión con datos

```
ConnectivityManager cm = (ConnectivityManager)
getApplicationContext().getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo activeNetwork = cm.getActiveNetworkInfo();

if (activeNetwork.getType() == ConnectivityManager.TYPE_MOBILE) {
    // Hay conexión de datos
    // Preguntar al usuario
}
```


PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Servicios en red III

Servicios de Google

Google Maps

Publicación de una aplicación en la Play Store

Utilización de servicios externos

Es posible **utilizar un servicio externo** e integrarlo en nuestras Apps Android, como ya hemos visto.

Una de las mayores reglas en el desarrollo de software nos dice que si algo ya está inventado y funciona, no lo reinventes y utilízalo.

En las aplicaciones móviles, y en las de escritorio también, podremos utilizar **servicios de terceros** que ya tienen implementados, por ejemplo:

- APIS de inicio de sesión con redes sociales,
- mapas de Google Maps,
- envío de mensajes en tiempo real con Firebase,
- etc.

Es aconsejable buscar un poco antes de intentar crear nuestro propio servicio, ya que puede estar creado o puede haber algo que nos ayude a terminar de implementarlo.

Google Maps

En **2003**, Lars y Jens Rasmussen, junto a los australianos **Noel Gordon y Stephen Ma**, fundaron la empresa Where 2 Technologies, con la que empezaron a mapear la ciudad de Sídney, en **Australia**.

En **2004**, **Google se interesó por esta idea** y **se la compró** a sus creadores. Así nació Google Maps. Se supone que fue este **año 2004** en el que **Google creó Google Maps**.

Oficialmente, Google Maps se anunció en **febrero de 2005** y lo soportarían los navegadores **Internet Explorer y Mozilla Firefox**, ya que, por aquel entonces, no existían los smartphones.

Google Maps utiliza un sistema de coordenadas mediante una latitud y una longitud, que son **positivas para el Norte y el Este** y negativas para el Sur y el Oeste.

El enlace a la web de Google Maps es el siguiente:

<https://www.google.es/maps/preview>

Google Maps no solo ofrece servicios en la Tierra, sino que también podremos disfrutar de unas breves zonas en la Luna, mediante **Google Moon**; y en Marte, mediante **Google Mars**.

El enlace a Google Moon es el siguiente:

<https://www.google.com/moon/>

El enlace a Google Mars es este:

<https://www.google.com/mars/>

Google nos ofrece la posibilidad de usar su API de Google Maps en nuestros dispositivos de una forma relativamente fácil y gratuita.

Complementos de los navegadores

Una cosa a tener en cuenta, es que al estar ofreciendo **servicios HTTPS** en nuestras aplicaciones, puede ser necesario el **visualizar algún sitio web** en nuestro dispositivo móvil.

Es importante saber que los navegadores utilizan ciertos **complementos para visualizar los sitios web** y que éstos pueden estar desactualizados en nuestros dispositivos móviles, así que no hay que olvidar tener todas las **apps y complementos** siempre **actualizados**.

Creación de un proyecto de Google Maps

Para utilizar la API de Google Maps en nuestras aplicaciones Android, debemos realizar algunos **pasos previos**.

En **primer lugar**, para hacer uso de todos los **servicios disponibles de Google Maps**, debemos generar una **Clave de API** (o **API Key**) para **asociarla** de forma unívoca a nuestra **aplicación** en concreto. Esto se hace muy rápido: se puede hacer accediendo a la **Consola de Desarrolladores** de Google desde nuestra cuenta de Gmail en el siguiente enlace y, posteriormente, siguiendo los pasos que mostramos seguidamente:

<https://console.developers.google.com/apis/dashboard>

1. Cuando accedamos, tenemos que **crear un proyecto** nuevo a partir de la lista desplegable que nos aparecerá arriba a la derecha. Para ello, seleccionamos la opción Crear proyecto.
2. Una vez hecho esto, se nos mostrará una nueva ventana que nos solicitará el **nombre del proyecto** que vamos a crear. Una vez hayamos introducido algún nombre descriptivo, se va a generar automáticamente un **ID único**. Solo nos queda terminar el proceso pulsando Crear.
3. Cuando tengamos creado nuestro proyecto, **lo seleccionamos** y pulsamos el enlace **biblioteca de API**, donde podremos seleccionar todas las API que nos ofrece Google y que utilizaremos. En nuestro caso particular, vamos a seleccionar **Maps SDK for Android**.
4. Ahora necesitaremos crear las **credenciales de nuestro proyecto**; para ello, pulsamos en Credenciales, que está a la izquierda. Una vez ingresemos en esta pantalla, pulsaremos en Crear credenciales y seleccionaremos **Clave de API**. Esto nos **creará la API** de nuestra aplicación, que permitirá a nuestras aplicaciones Android **usar los mapas** de Google.

Deberemos **guardar la clave de API** que se genere para utilizarla más adelante, aunque siempre podremos **acceder** a ella desde nuestra **cuenta y proyecto**.

Una vez hecho todo esto, tendremos listo nuestro proyecto de Google Maps para utilizarlo desde nuestra aplicación Android.

Cupo de proyectos

Tenemos que tener cuidado cuando creamos **proyectos de Google**, ya que sólo tendremos un cupo aproximado de **19 proyectos de forma gratuita**.

Cuando estamos desarrollando aplicaciones, es relativamente sencillo empezar a crear proyectos sin control para probar servicios, y posteriormente, vernos en la necesidad de tener que eliminar algunos de ellos.

Los registros como desarrollador

Cuando vamos a utilizar servicios de este tipo que ofrece un tercero, es muy normal que debamos registrarnos para obtener un ID de desarrollador. Esta práctica se da **por seguridad** y para poder tener **controlado el uso de los servicios** por parte de la compañía que los proporciona, ya que es un aspecto bastante serio y está **regulado por ley**.

El registro como desarrollador lo vamos a encontrar en todas las grandes empresas que ofrecen servicios de este tipo, como pueden ser Google, para utilizar sus servicios de **Google Maps o YouTube, Facebook** (para poder usar el inicio de sesión a través de su plataforma), **Twitter**, etc.

Configuración de un proyecto en Android Studio para utilizar Google Maps

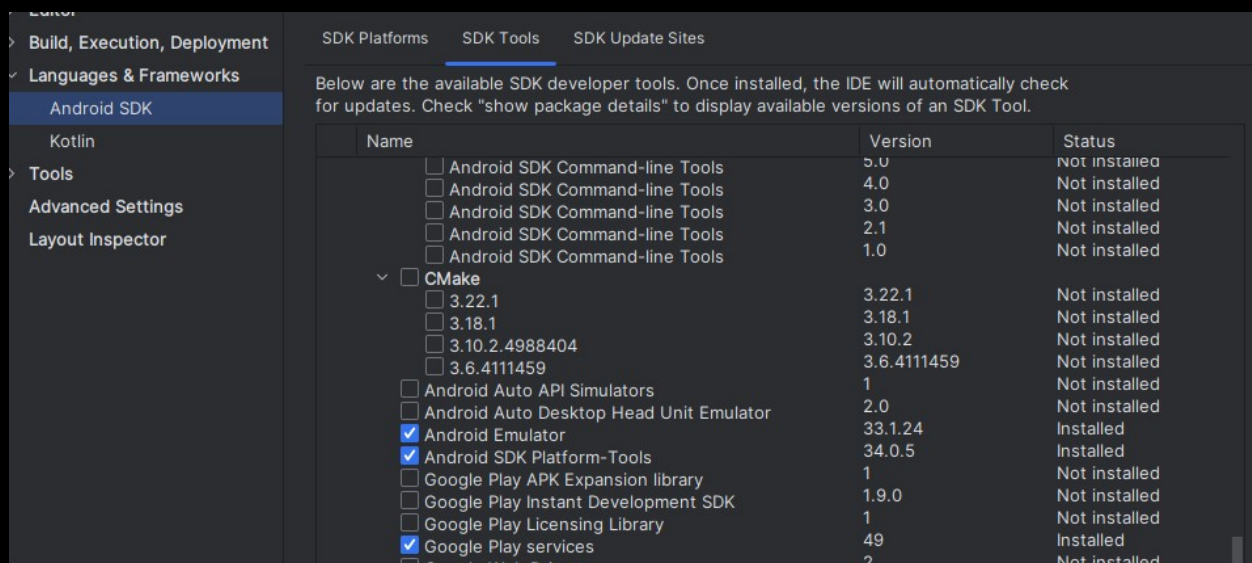
Una vez tenemos configurado nuestro proyecto de Google Maps en la consola de desarrolladores de Google, con la clave de API que nos permitirá su uso, estamos en condiciones de **crear un proyecto en Android Studio** para poder utilizar el servicio de Google Maps.

Lo primero que debemos hacer es **instalar** (si no la tenemos) la **SDK de Google Play Services** en nuestro Android Studio. Este es un paso importante, ya que, sin esta SDK, no aparecerán los mapas en las aplicaciones del emulador.

Para ello, nos vamos al **gestor de SDK de Android Studio (SDK Manager)** y, en la opción **Android SDK**, seleccionamos SDK Tools.

Aquí nos aparecerá la SDK de **Google Play Services** para instalar (si no está ya instalada).

Una vez hecho esto, estará **instalado para todos nuestros proyectos**, ya que se instala en la SDK de Android Studio.




A continuación, en nuestro proyecto deberemos poner las **dependencias de Google Maps** en el fichero **build.gradle** de la siguiente forma:

```
implementation("com.google.android.gms:play-services-maps:18.2.0")
```

Nota: Android Studio marcará que no está actualizada la versión del SDK, deberemos ir a:

https://play.google.com/sdks/details/com-google-android-gms-play-services-maps?utm_source=ide&utm_medium=snapshot

Y en “Versiones del SDK” ver cual es la última versión y cambiar a esa versión en la implementación de arriba. Entonces la advertencia subrayada en rojo se quitará.

Versiones del SDK ⓘ		
Versión	Uso	
18.2.0	6 %	<div><div></div></div>  Última versión
18.1.0	27 %	<div><div></div></div>

Finalmente, modificaremos el fichero **AndroidManifest.xml** para añadir estos tres elementos nuevos:

Nota: gms.version y geo.API_KEY se agregarán dentro del elemento <application> ... </application>.

Mientras que GLESVersion se agregará como hijo directo de <manifest>, quedando así la estructura:

Elementos nuevos por añadir

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

  <uses-feature
    android:glEsVersion="0x00020000"
    android:required="true" />

  <application
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
```

```
android:roundIcon="@mipmap/ic_launcher_round"
android:supportsRtl="true"
android:theme="@style/Theme.TestGoogleMapsMedac"
tools:targetApi="31">
<meta-data
    android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="AIzaSyCO2L1f5ldqVvQlgeUUzJEAs2LKlk4Ep8" />
```

[...]

Ayuda:

<https://developers.google.com/maps/documentation/android-sdk/start?hl=es-419>

<https://developer.android.com/guide/topics/manifest/uses-feature-element?hl=es-419>

<https://developers.google.com/maps/documentation/android-sdk/config?hl=es-419#kotlin>

Agregar un mapa de Google Maps

Para agregar un mapa de Google Maps a nuestra actividad, simplemente tenemos que agregar el **control al Layout** de la actividad donde queramos mostrar el mapa.

Para poder añadirlo, necesitaremos agregar un fragment a la actividad de tipo `com.google.android.gms.maps.MapFragment`, el cual quedará como se muestra a continuación.

Fragment con un mapa

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

    <fragment xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/map"
        android:name="com.google.android.gms.maps.MapFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</LinearLayout>
```

Puede ocurrir que no se muestre correctamente, pero puede deberse a algún error de renderizado en el editor visual de Layouts de Android Studio. Con cerrar la pantalla y volverla abrir, se solucionará.

Para poder **obtener la referencia al mapa** en nuestra actividad, vamos a crear, en primer lugar, un **atributo privado** de tipo `GoogleMap` en la actividad principal. Haremos, además, que nuestra actividad **implemente** la interfaz **`OnMapReadyCallback`**, lo cual nos permitirá **dar funcionalidad** a nuestro mapa.

Debemos modificar el método `onCreate()` de la actividad, obteniendo, en primer lugar, una **referencia al `MapFragment`** que hemos añadido a nuestro Layout a través del fragment manager; seguidamente, llamaremos a su método **`getMapAsync()`**, pasándole un **objeto que implemente la interfaz `OnMapReadyCallback`**, en nuestro caso la **propia actividad principal**.

Con esto, conseguimos que, en cuanto esté disponible la referencia al mapa incluido dentro de nuestro MapFragment, se llame automáticamente al método **onMapReady()** de la interfaz. Por el momento, la implementación de este método va a ser muy sencilla, ya que nos limitaremos a guardar el objeto GoogleMap recibido como parámetro en nuestro atributo mapa.

Una vez hecho esto, podremos ejecutar nuestra aplicación y tendremos un mapa funcional.

En el siguiente vídeo, podrás visualizar el proceso completo, así como la opción de cambiar de mapa. En la sección de «Recursos del tema» encontrarás todo el código generado.

Integración de un mapa de Google en nuestra aplicación cambiando el tipo de mapa

En primer lugar deberemos abrir el fichero `build.gradle` del módulo aplicación, para colocar esta implementación que es la biblioteca de Google Maps.

```
implementation 'com.google.android.gms:play-services-maps:18.2.0'
```

Una vez hagamos esto aquí nos aparecerá una opción de **sincronización** de nuestro proyecto, la pulsamos y se descargará la biblioteca para poder usarla.

Luego, en el **Manifest** deberemos de poner las siguientes configuraciones:

- `uses-feature` que deberá ir justo encima de `application`.
- Los dos elementos meta-data vistos antes, que uno indica nuestro número de Google Play Services y el otro indica nuestro código de la aplicación de Google Maps en la nube.

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

    <uses-feature
        android:glEsVersion="0x00020000"
        android:required="true" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <meta-data
            android:name="com.google.android.gms.version"
```

```

        android:value="@integer/google_play_services_version" />

<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="AlzaSyCO2Ll1f5ldqVvQlgeUUzJEAs2LKlk4Ep8" />

<activity
    android:name=".MainActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

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

</manifest>

```

Ahora vamos a ver la interfaz gráfica:

Podemos ver que tenemos:

- un `fragment` con un mapa,
- un `RadioGroup` con cuatro `RadioButtons`. Estos serán los cuatro tipos de mapas que podremos poner. Pulsando sobre cada uno se cambiará el tipo de mapa mostrado, y se descargará dicho mapa.

En nuestra aplicación, en la clase `MainActivity`, deberemos de:

1. crear un objeto de tipo `Google Maps` y otro objeto `MapFragment` a los que haremos un `findViewById`.
2. implementar las interfaces `OnMapReadyCallback` y `GoogleMap.OnMapClickListener`, además de la del `RadioButton`, para saber qué `RadioButton` está activo.

Clase MainActivity del proyecto

```
public class MainActivity extends AppCompatActivity implements OnMapReadyCallback,
RadioGroup.OnCheckedChangeListener, GoogleMap.OnMapClickListener{

    private GoogleMap mapa;
    private MapFragment mapFragment;
    private RadioGroup rgTipoMapa;

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

        // ***** Ligamos los recursos de la actividad *****
        mapFragment = (MapFragment) getFragmentManager().findFragmentById(R.id.map);
        rgTipoMapa = (RadioGroup) findViewById(R.id.rgTipoMapa);
        // *****

        mapFragment.getMapAsync(this);
        rgTipoMapa.setOnCheckedChangeListener(this);
    }
}
```

En el método OnMapReady:

Deberemos **igualar nuestro** mapa al mapa que ya está **creado**, que es el que le llega, y deberemos indicar que el **listener** del click es **nuestra propia clase (this)**.

```
@Override
public void onMapReady(GoogleMap map)
{
    mapa = map;
    mapa.setOnMapClickListener(this);
}
```

Cuando cambiemos y pulsemos sobre los RadioButton haremos lo siguiente:

- Para cambiar a una vista normal, con el método `SetMapType` deberemos pasar la variable `MAP_TYPE_NORMAL`.
- Para una vista en satélite `MAP_TYPE_SATELLITE`.
- Para una vista en híbrido que será una combinación de normal y satélite será `MAP_TYPE_HYBRID`.
- Y para una vista **topográfica** será `MAP_TYPE_TERRAIN`.

Todas ellas son variables finales de la clase `Google.Map`.

```
@Override
public void onCheckedChanged(RadioGroup radioGroup, @IdRes int checkedId)
{
    switch(radioGroup.getId())
    {
        case R.id.rgTipoMapa:
            switch(checkedId)
            {
                case R.id.rbNormal:
                    mapa.setMapType(GoogleMap.MAP_TYPE_NORMAL);
                    break;
                case R.id.rbSatelite:
                    mapa.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
                    break;
                case R.id.rbHibrido:
                    mapa.setMapType(GoogleMap.MAP_TYPE_HYBRID);
                    break;
                case R.id.rbTopografico:
                    mapa.setMapType(GoogleMap.MAP_TYPE_TERRAIN);
                    break;
            }
            break;
    }
}

@Override
public void onMapClick(LatLng latLng) {

}
}
```

Agregar marcadores en Google Maps

Una de las acciones más comunes que podemos realizar en Google Maps es usar marcadores. Con los marcadores, podremos **dejar seleccionada una ubicación** en el mapa, con la que podremos interactuar.

Las coordenadas que usa Google Maps se basan en una latitud y en una longitud, que deberemos conocer para poder indicarle que coloque un marcador en dicha ubicación.

Si quieres averiguar las coordenadas GPS de cualquier lugar del mundo, pincha en el siguiente enlace:

<http://www.coordenadas-gps.com/>

Para **responder a los clics del usuario** sobre el mapa, definiremos el **evento** `onMapClick()`, llamando al **método** `setOnMapClickListener()` de nuestro mapa.

Dentro de este, recibiremos un **objeto** `LatLng` con la **posición geográfica**.

Para **obtener esta posición** usaremos el método `toScreenLocation()` del **objeto** `Projection` que podemos **obtener a partir del mapa** llamando a `getProjection()`.

Una vez sabemos cómo podemos obtener la posición, podremos **agregar un marcador** a dicha posición, mediante el método `addMarker`, al que le deberemos pasar un objeto del tipo `LatLng` con las **coordenadas** y un **título** para identificar el marcador al pulsar en él. También podremos agregar un marcador en el lugar que deseemos, utilizando la página web que nos proporciona las coordenadas de una dirección cualquiera.

En el siguiente vídeo, podrás visualizar. En la sección de «Recursos del tema» encontrarás todo el código generado.

Cómo se agregan marcadores, paso a paso

Vamos a ver cómo podemos agregar marcadores a un mapa de Google.

Una vez que ya hemos **configurado nuestra aplicación** para que pueda utilizar la biblioteca de Google Maps y esté configurado el **fichero Manifest**, en nuestra interfaz gráfica tendremos lo siguiente:

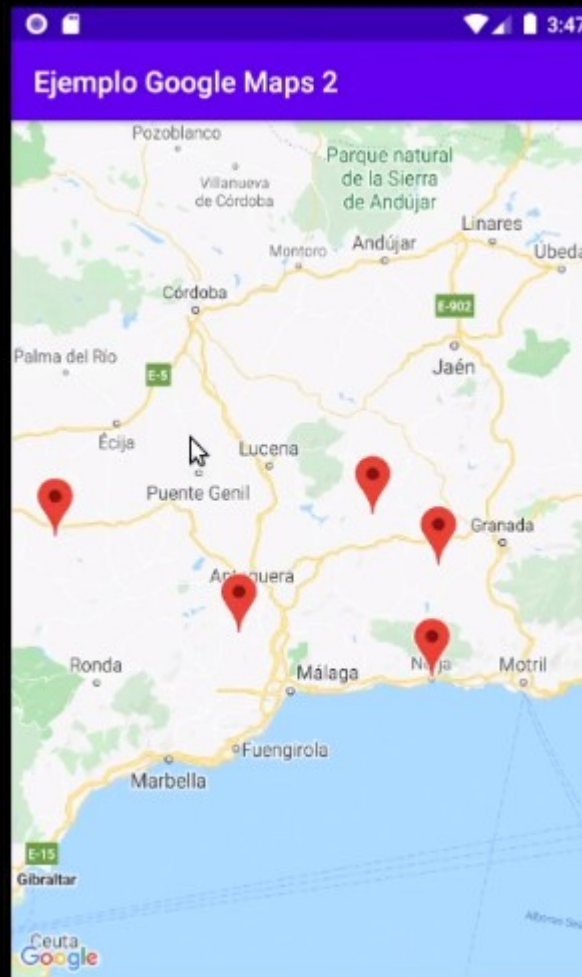
Un mapa de Google en el que lo que vamos a hacer es que cuando pulsemos sobre una localización se agregue automáticamente un marcador. Para ello:

En la clase `ActivityMain` haremos lo siguiente:

Crearemos un **Google Map** y un **MapFragment**. Hemos creado un **método** `insertarMarcador` que inserta un marcador en un punto dado con una latitud y una longitud las cuales son sus coordenadas. Para ello vamos a utilizar el **método** `addMarker` al que le pasaremos la posición en coordenadas y un título, que siempre será en este caso "Marcador". Como hemos implementado la **interfaz** `onMapClickListener` de `GoogleMaps`, usaremos el

método sobreescrito **onMapClick** que hará que cuando hagamos clic sobre el mapa se ejecute una acción. Esa acción será la de **insertar el marcador** ya que cuando hagamos clic, por defecto se nos dará la latitud y longitud de donde hemos pulsado.

Vamos a probar nuestra aplicación. Aquí tenemos un mapa de Google que haciendo doble clic se nos da la latitud y longitud de donde hemos pulsado. Con un simple clic podremos acercarnos a una localización, por ejemplo a España. Ahora si hacemos clic sobre una localización **se agregará un marcador** donde hemos hecho el clic. Podremos agregar tantos marcadores como queramos.



Si pulsamos sobre un marcador seguirá el foco a dicho marcador y aparecerá el **título**. En este caso todos tendrán el mismo título.



MainActivity implementando la interfaz onMapClickListener

```
public class MainActivity extends AppCompatActivity implements OnMapReadyCallback,
    GoogleMap.OnMapClickListener {

    private GoogleMap mapa;
    private MapFragment mapFragment;

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

        // **** Ligamos los recursos de la actividad ****
        mapFragment = (MapFragment) getFragmentManager().findFragmentById(R.id.map);
        // ****

        mapFragment.getMapAsync(this);
    }

    @Override
    public void onMapReady(GoogleMap map)
    {
        mapa = map;
        mapa.setOnMapClickListener(this);
    }

    @Override
    public void onMapClick(LatLng latLng)
    {
        insertarMarcador(latLng);
    }

    /**
     * Inserta un marcado en un Google Maps
     * @param coordenadas Coordenadas para insertar dicho marcador
     */
    private void insertarMarcador(LatLng coordenadas)
    {
        mapa.addMarker(new MarkerOptions()
            .position(coordenadas)
            .title("Marcador"));
    }
}
```

Nuevos servicios

Enviar mensajes instantáneos entre dispositivos móviles.

Uno de los servicios más amplios es el de **FireBase**. Con este servicio, podremos **enviar mensajes** entre diferentes dispositivos móviles, como lo hacen las aplicaciones de WhatsApp, Telegram, Messenger, etc. Podremos crear **bases de datos en tiempo real** en la nube, así como otras operaciones también en tiempo real, y muchos más servicios que pueden hacer que nuestras aplicaciones crezcan de forma exponencial.

Por ejemplo, mediante Firebase, podremos enviar mensajes de **texto y multimedia**, con imágenes, entre usuarios, tal y como podemos hacer con aplicaciones como WhatsApp o Telegram. Aquí deberíamos **dar permisos a nuestra aplicación**, tanto para poder **conocer el ID** de nuestro dispositivo como para poder **acceder a las fotografías** del mismo, en el caso de enviar mensajes multimedia.

Página web:

<https://firebase.google.com/?hl=es>

Huawei, información y entornos de pruebas:

Push: <https://developer.huawei.com/consumer/en/hms/huawei-pushkit/>

Analytics: <https://developer.huawei.com/consumer/en/hms/huawei-analyticskit/>

Location: <https://developer.huawei.com/consumer/en/hms/huawei-locationkit/>

Maps: <https://developer.huawei.com/consumer/en/hms/huawei-MapKit/>

Anuncios: <https://developer.huawei.com/consumer/en/hms/huawei-adskit>

Account: <https://developer.huawei.com/consumer/en/hms/huawei-accountkit>

Publicación de una aplicación en la Play Store de Google

Para poder publicar nuestras aplicaciones en Google Play, deberemos registrarnos en la **Consola para Desarrolladores de Google**. Esto supone un coste de unos **25 dólares** que deberemos pagar mediante tarjeta de crédito y tendremos una **licencia de por vida** para poder subir todas las aplicaciones que queramos. Para esto, solo necesitamos una **cuenta en Gmail** y acceder a la siguiente página web, para registrarnos y **realizar el pago**:

<https://play.google.com/apps/publish>

Tendremos que seguir **cuatro pasos**:

1. Iniciar sesión con nuestra **cuenta de Gmail**, la cual quedará vinculada y **será la cuenta de desarrollador**.
2. **Aceptar** el acuerdo.
3. Realizar el **pago**.
4. Rellenar la **información** pertinente.

Para poder subir las aplicaciones, deberemos **crear un APK** desde Android Studio. Previamente, deberemos **crearnos una firma** para nuestras aplicaciones, lo cual podremos hacer también desde Android Studio. Para esto último:

<https://developer.android.com/studio/publish/app-signing?hl=es-419>

1. Pulsaremos en el **menú Run** (menú Build en actual versión Hedgehog -Erizo- | 2023.1.1), y **Generate Signed Bundle/APK**. Elegiremos la **opción APK** y, en la siguiente pantalla, podremos crear una Key Store o firma pulsando en **Create New**. **Si ya disponemos** de una, pulsamos **Choose existing**. Esta firma nos identificará como el propietario de la aplicación (estas solo se podrán subir si están firmadas).

New Key Store

Key store path: ~/user/keystores/upload-keystore.jks

Password: Confirm:

Key

Alias: upload

Password: Confirm:

Validity (years): 25

Certificate

First and Last Name: First Last

Organizational Unit: Mobile

Organization: MyCompany

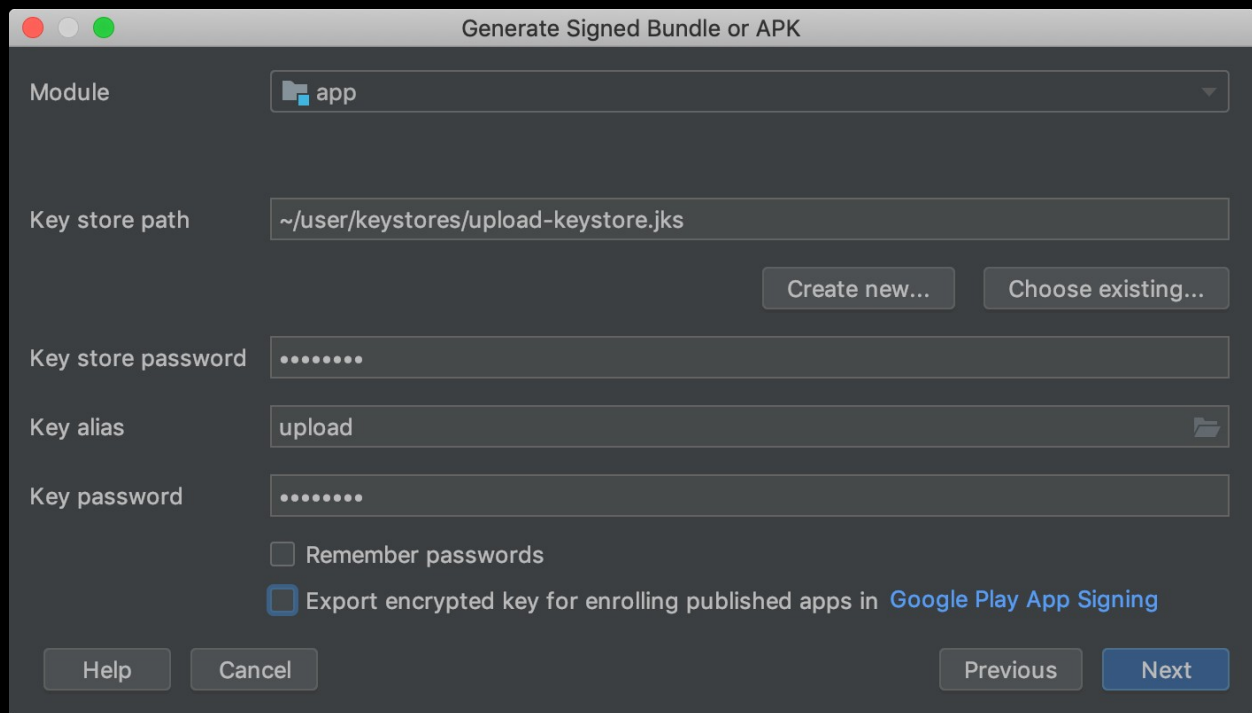
City or Locality: MyCity

State or Province: MyState

Country Code (XX): US

Cancel OK

(Crea una clave de carga y un almacén de claves nuevos en Android Studio)



(Firma la app con la clave de carga)

2. Terminamos de **crear nuestra APK** y, una vez la tengamos, nos dirigimos a la [Consola de Desarrollador](#). Pulsaremos en **Crear una aplicación** y nos aparecerá una pantalla para elegir el **idioma** por defecto y el **título** de la misma.
3. Una vez **creemos la aplicación**, podremos **subir nuestro APK**, el cual pasará a revisión por parte del equipo de Google y, si todo está bien, se publicará en unas horas.
4. Dentro de la [configuración de nuestra aplicación](#), podremos **traducirla** a otros idiomas, subir **capturas** de pantalla, subir el **icono** de la misma, etc.

Podremos tener todas las aplicaciones que queramos, siempre y cuando no infrinjan ninguna ley actual.

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Introducción a los videojuegos

Motor de videojuego

Los videojuegos. Concepto

Los videojuegos son mundialmente conocidos, pero ¿cómo podríamos definir qué es un videojuego? Podemos decir que un videojuego es un **mecanismo electrónico capaz de interactuar con uno o varios jugadores** para poder conseguir una serie de **objetivos, o misiones**, proporcionando una respuesta a cada acción del jugador.

Los videojuegos se pueden **clasificar mediante plataformas**, que son en sí el sistema donde se ejecutan los mismos.

Pueden ser:

- PC.
- PlayStation 1/2/3/4/5.
- Xbox.
- Dispositivos Móviles.
- Consolas de Nintendo (Switch, GameBoy, etc.).
- Máquinas recreativas.

Los videojuegos pueden ser:

- Desarrollados para **varias plataformas**, como, por ejemplo, Dark Souls Remastered, que está disponible para Nintendo Switch, PlayStation 4, Xbox One y PC.
- **Exclusivos**, es decir, que solo están **disponibles en una plataforma**, por ejemplo, Bloodborne, que únicamente está disponible para PlayStation 4.

La **arquitectura global** de un videojuego es una construcción que la componen distintos **bloques funcionales**, de los que destacamos:

- **Historia**: Es la manera en que se desenvolverán los personajes del juego y el hilo conductor del mundo que se representará.
- **Arte conceptual**: Mostrará el **aspecto general** del juego.
- **Sonido**: Voces, sonidos ambientales, efectos sonoros y música.
- **Mecánica de juego**: Donde se especificará el **funcionamiento general** del juego.
- **Diseño de programación**: Mediante este elemento se describirá la manera en que el videojuego será **implementado** en una máquina (PC, consola, móviles, etc.) mediante un determinado **lenguaje** de programación y siguiendo una **metodología** concreta.

Historia de los videojuegos

Los primeros videojuegos se remontan sobre el año **1950**, al poco tiempo de la creación de las primeras computadoras electrónicas. Todos los videojuegos que se realizaron en esta época eran muy rudimentarios, incluso algunos no podrían ser catalogados como videojuegos en sí.

Sobre el año **1970** apareció una compañía que lo cambiaría todo, esta fue **Atari**, que con su videojuego **Pong**. Crearon lo que hoy se conoce como **máquinas recreativas**, iniciando una fortísima revolución en el mundo de los videojuegos.

A partir de este momento la industria del videojuego fue en aumento, llegando en los **años 80** las **videoconsolas con 16 bits**.

Otros hitos de este sector fueron la creación de la **primera PlayStation** por parte de Sony en el año **1994**, la salida al mercado de la **GameBoy de Nintendo** en **1986**, la aparición de la **XBOX** de Microsoft en el año **2002**, etc.

Clasificación de los videojuegos

Dentro del mundo de los videojuegos, existe una gran cantidad de clasificaciones posibles. Además de la clasificación por plataformas comentada anteriormente, podemos destacar:

- Videojuegos de **acción**: Este tipo de videojuegos son aquellos en los que habrá que realizar unos movimientos y combates rápidos.
- Videojuegos de **aventura**: Son aquellos en los que hay una **historia compleja**, que deberá ir desarrollando poco a poco el jugador. Algo muy típico en este tipo de videojuegos es la inclusión de algún tipo de puzle para hacer que el jugador tenga que pensar cómo resolverlo.
- Videojuegos de **lucha**: Consisten en pelear o luchar contra otros personajes del videojuego, (los llamados **NPC, Non-Player Character**. Personajes No Jugables en español) o contra otros jugadores en un modo multijugador.
- Videojuegos de **plataformas**: Consisten en que el jugador deberá ir **moviéndose y saltando sobre una serie de terrenos** o plataformas para avanzar en el videojuego. Quizás, el más famoso de esta categoría sea Super Mario Bros.
- Videojuegos de **estrategia**: Estos requerirán al jugador una **planificación** bastante elevada de sus movimientos, ya que de ello dependerá su victoria.
- Videojuegos **puzles**: Son los que ofrecen una serie de puzles al jugador donde este deberá averiguar cómo poder resolverlos para ir avanzando en el mismo.
- **Simuladores**: Son los que ofrecen una simulación de tareas del mundo real. Pueden ser simuladores de vuelo, de negocios, etc.

En el mundo de los videojuegos, es habitual que haya modificaciones de los mismos mediante **actualizaciones o parches** que proporcionará el desarrollador, ya sea para introducir alguna **mejora** o por adaptarse **mejor al tipo de mercado** que lo demanda. Incluso, es muy común que se realicen

remakes completos de un videojuego antiguo e incluir en el mismo nuevas zonas de mapeado, personajes, misiones, etc. Este es el caso de Demon Souls en su remake para PlayStation5.

Una ventaja de desarrollar videojuegos para videoconsolas frente a PC es que su hardware no se puede ampliar.

¿Por qué el hecho de no poder ampliar el hardware constituye una ventaja?

A pesar de que pueda parecer una ventaja el hecho de poder ampliar el hardware realmente no lo es, sobre todo para los desarrolladores, ya que si el hardware tiene un tamaño fijo, los desarrolladores ya conocen de antemano exactamente las especificaciones del hardware para el cual están creando el juego. Entonces no hay variabilidad en términos de tarjetas gráficas, procesadores o memoria RAM como puede haber en PC. Esto es una ventaja ya que permite una mayor estabilidad y consistencia en el rendimiento del juego.

El por qué de los motores de videojuegos

Los videojuegos se componen de multitud de bloques, y no todos son sencillos de implementar, por ejemplo, ¿cómo podemos hacer que salte un personaje?, ¿y si queremos que uno de los enemigos del videojuego persiga a nuestro personaje?

Todas estas partes se pueden hacer desde cero, pero no es nada recomendable, ya que si investigamos un poco, podremos observar que existe un componente llamado **motor de un videojuego**, que ya nos **implementa las bases** de la gran mayoría de **elementos** que vamos a necesitar realizar en nuestros videojuegos. La implementación de cero de un motor completo puede ser una tarea colosal, así que siempre optaremos por **usar un motor que ya esté implementado** y disponible.

Para conocer algunos de los entornos de desarrollo básicos que te permitirán iniciarte en la creación de videojuegos sin que intervenga demasiado la programación, puedes consultar el siguiente [enlace](#).

Motores de videojuegos

Los videojuegos de hoy en día necesitan una gran cantidad de recursos para poder ejecutarse, basta con ver los requerimientos mínimos que necesita cualquier videojuego de PC o las características de las videoconsolas actuales.

En los videojuegos, podemos distinguir **tres grandes partes**:

1. El código: Este será todo el código que se necesitará para crear el juego en sí. En esta parte será donde se **implementará toda la lógica** del juego.
2. El motor del juego.
3. Los recursos requeridos: Estos son los recursos necesarios por el juego, como pueden ser banda sonora, sonidos de efectos, imágenes, etc.

Quizás, la **parte más importante** de todo videojuego sea su **motor**, ya que este **va a contener la funcionalidad** sobre la que se va a apoyar directamente el videojuego.

Los motores de los videojuegos surgieron como tales en los **años 90** y se empezaron a utilizar en los **videojuegos 3D**, ya que estos necesitaban una potencia de cálculo muy elevada. **También** podemos usar motores en los **videojuegos 2D**, aunque estos serán mucho más simples que los utilizados en los 3D.

Los motores nos van a proporcionar una **abstracción sobre el hardware** del dispositivo, lo cual va a permitir a los desarrolladores del videojuego poder **programar sin tener la necesidad de conocer la arquitectura del hardware** sobre el que se ejecutará el videojuego. Debido a esto, la gran mayoría de los motores existentes se han **desarrollado a partir de una API ya implementada**, como puede ser OpenGL, DirectX, SDL, etc.

Otra gran ventaja de esta abstracción del hardware es que puede **permitir** el desarrollo de **aplicaciones multiplataforma**.

Es importante mencionar que con los **primeros videojuegos** se utilizaba una **aceleración de gráficos 3D basada en hardware**, lo cual implicaba que el **hardware** de la máquina donde se ejecutaría el videojuego **debía ser bastante potente** para soportar dicha aceleración.

Clasificaciones de los motores de videojuegos

<https://bit.ly/36RLnYe>

Según el nivel de abstracción

(Tipo de motor : Descripción)

Motores para mostrar en pantalla:

Este tipo de motores son **muy simples**, y están diseñados únicamente para mostrar elementos por pantalla. Como ejemplos tenemos: Biblioteca **SDL**.

Motores con soporte de lógica:

Este tipo motores agregan a lo anterior un **soporte total o parcial de la lógica** del videojuego. Como ejemplos tenemos: **Ogre3D**.

Motores que incluyen plataformas para creación de contenido:

Este tipo de motores agregan a lo anterior una serie de programas que permitirán **crear contenido** para los videojuegos. Como ejemplos tenemos: **Blender**.

Según la licencia de uso

(Tipo de motor : Descripción)

De código abierto:

Para poder usar este tipo de motores no hace falta pagar, sólo hay que prestar atención al tipo de **licencia** de código abierto que esta usando y **respetarla**. Como ejemplos tenemos: **Ogre3D**.

Propietarios:

Estos motores son creados por grandes empresas y su uso estará limitado mediante una **licencia de uso**, la cual costara dinero. Como ejemplos tenemos: **UnrealEngine**.

Explicado:

Clasificaciones de Motores de Videojuegos:

Vamos a ver las diferentes clasificaciones de los motores de videojuegos. Podemos hacer dos principales distinciones.

1. Nivel de Abstracción:

- Motores para mostrar elementos por pantalla: Este tipo de motores son muy simples y se encargan únicamente de mostrar cosas por pantalla, como texto, imágenes o polígonos. Un ejemplo común es la biblioteca SDL, utilizada en el desarrollo del videojuego Age of Empires 2.
- Motores con soporte de lógica: Agregan a la funcionalidad de mostrar elementos por pantalla un soporte total o parcial de la lógica del videojuego. Un ejemplo de este tipo de motor es Ogre 3D.
- Motores que incluyen plataformas para creación de contenido: Estos motores, además de contar con soporte de lógica, ofrecen la posibilidad de crear contenido, como modelado de personajes, enemigos o escenarios. Blender es un ejemplo que permite desarrollar videojuegos y diseñar modelos 3D de personajes.

2. Licencia de Uso:

- Motores de código abierto: Permiten descargar su código y su uso es gratuito.
- Motores propietarios: Desarrollados por empresas propietarias que no proporcionan su código fuente y suelen requerir el pago de licencias.

En resumen, podemos encontrar cualquiera de estos tres tipos de motores con cualquiera de estas dos licencias de uso.

Ventajas de los motores de los videojuegos

Ya sabemos que los motores en los videojuegos van a soportar gran parte del peso de la lógica de los mismos, y es por eso que su uso trae consigo múltiples ventajas.

En la siguiente tabla, podemos ver **algunas de las más notables ventajas**, según diferentes situaciones que se pueden presentar:

Comparación con y sin motor

Migración del videojuego a otra plataforma distinta:

- **Sin usar motor:** No queda más remedio que **revisar el código completo** del videojuego y ver qué partes deberemos cambiar para poder migrarlo a la plataforma deseada. Puede ocurrir que haya recursos que ya no sean válidos en la nueva plataforma. Si llegamos a detectar **errores** en el código del videojuego, habrá que **cambiarlos en todas las plataformas** que tengamos.
- **Usando motor:** Si el motor que estamos usando es soportado por la nueva plataforma, lo único que tendremos que hacer será **recompilar de nuevo el proyecto** del videojuego.

Llegada de nuevos programadores al proyecto:

- **Sin usar motor:** Habrá que **explicarles** a los nuevos programadores toda la **estructura del código** del proyecto.
- **Usando motor:** Lo único que habrá que hacer será **formar** a estos nuevos programadores en el **uso del motor** utilizado.

Incorporación de nuevos efectos a nuestro videojuego:

- **Sin usar motor:** Habrá que invertir tiempo en **investigar cómo funciona** la característica nueva que queremos **incorporar, implementarla y realizar todas las pruebas** pertinentes para asegurar su funcionamiento. Además, habrá que hacerlo en todas las plataformas de las que dispongamos.
- **Usando motor:** Seguramente, el **motor** que estemos usando **implemente la característica** que queremos agregar.

Realizar una segunda parte de un videojuego:

- **Sin usar motor:** Tendremos que **volver a revisar todo el código** del mismo para ver qué elementos podremos reutilizar en la siguiente parte y cuáles no.
- **Usando motor:** Aquí no habrá ningún tipo de problema, ya que el **código está separado del motor**.

Como observamos, las ventajas de usar un motor en el desarrollo de videojuegos pueden llegar a ser muy importantes.

Estas son solo algunas de ellas, por lo que seguro que en el día a día de un proyecto se presentarán más situaciones que podremos solventar fácilmente si se están usando los motores.

Sin embargo, habrá que indicarles ciertos comportamientos para que puedan procesar los modelos.

Motores 2D

Los videojuegos en 2D son aquellos que solamente utilizan **2 dimensiones**, alto y ancho, dejando sin utilizar la profundidad. Este tipo de juegos utilizan **formas planas** que se desplazan por la pantalla, hacia los lados, y hacia arriba y abajo.

Los motores 2D nos van a permitir **dibujar de una forma más sencilla las figuras geométricas**, como pueden ser círculos, cuadrados, rectángulos, triángulos, etc., sobre las que estarán basados todos los componentes de cada acción.

Este tipo de videojuegos tienen **varias capas** en las que en cada una se irá representando **una parte del juego**, como pueden ser los personajes, los enemigos, las monedas, etc.

Los llamados **sprites** (significa duendecillo en inglés) serán las **representaciones de los elementos** del juego (personajes, enemigos...) en **cada una de las capas**. Estos sprites **van a reproducir todos los movimientos** que puedan realizar los personajes o, en definitiva, cualquier elementos del videojuego.

El encargado de dar el **soporte** necesario **a los sprites** será el **motor 2D**, el cual tendrá herramientas que nos van a permitir **escalarlos, rotarlos, “recortarlos”** para obtener un fragmento donde estará el elemento que queremos mostrar en pantalla, etc.

Los sprites se irán superponiendo respetando las capas del videojuego, por ejemplo, primero se dibujará el fondo, luego se dibujará al personaje con el que jugaremos, posteriormente los enemigos y las partes del mapeado donde podremos interactuar, y por último, se dibujará la información del mismo, como pueden ser la cantidad de vidas que tenemos, el tiempo que llevamos jugando, nuestra puntuación, etc.

Ejemplo de videojuego en 2D: Plants vs Zombies.

Fuente: <https://www.pngocean.com/gratis-png-clipart-xihxz/descargar>

Concepto de Colisión en Videojuegos en 2D:

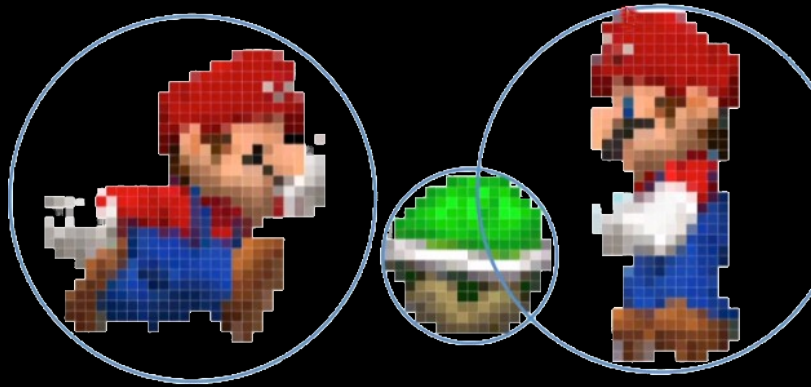
Vamos a ver qué es el concepto de colisión. Como ya sabemos, los videojuegos en dos dimensiones utilizan sprites para representar los personajes. Cuando utilizamos diferentes personajes, tendremos varios sprites moviéndose por el escenario entre comillas.

Puede ocurrir que en cierto momento haya dos sprites que se toquen o se crucen dentro del escenario, y a eso es a lo que llamamos una **colisión**. Las colisiones pueden darse entre personajes de jugadores, personajes de jugadores y personajes enemigos, o incluso con elementos del entorno de nuestro videojuego.

Lo más común para determinar si un enemigo ha matado a un personaje en un videojuego en 2D es calcular si ha habido una colisión entre ellos. En tal caso, podremos hacer que el jugador pierda la partida o ir quitándole vida al jugador.

Detección de Colisiones:

Para la detección de colisiones, hay varios métodos. Como podemos ver en esta imagen, tenemos tres sprites: uno, dos y tres. Se ha colocado una circunferencia sobre cada sprite, que es el método más preciso para la detección de una colisión. La circunferencia representa el espacio que ocupa nuestro personaje dentro del espacio de nuestro videojuego.



También es posible realizar la detección mediante cuadrados, aunque este método no es tan preciso como el de la circunferencia. Trozos como el mostrado aquí, por ejemplo, podrían provocar una colisión cuando en realidad no la hay.

Para determinar si ha habido una colisión, debemos observar dónde están los espacios ocupados por nuestros sprites. Si alguno de ellos se cruza, como en este caso, tendremos una colisión entre esos dos sprites.

La detección de colisiones es responsabilidad del motor 2D, motor 5D, motor 6D, motor 7D y motor 8D.

¿Cuál es el primer paso para comenzar con el desarrollo de mi videojuego?

Es común sentirse perdido al iniciar en el mundo del desarrollo de videojuegos, incluso cuando se trata de proyectos simples. La pregunta inicial podría ser: ¿deberíamos comenzar a programar y ajustar según los resultados, o sería más adecuado elaborar un pequeño guión primero? A menudo, entendemos que el proceso de crear un videojuego, como los que vemos en el mercado actual, implica diversas etapas:

- La **primera**, aunque obvia, es concebir una **buena idea** para el juego, algo atractivo y cautivador para los jugadores.
- Después de tener la idea, es crucial no lanzarse a programar sin un plan claro. Es necesario realizar un estudio exhaustivo sobre el tema central del videojuego y plasmarlo en un guión que defina los pasos a seguir en el desarrollo. Identificar los componentes clave, establecer un orden para su desarrollo, reconocer las herramientas de apoyo disponibles, decidir la plataforma objetivo (ya que no es lo mismo desarrollar para PC, PlayStation o dispositivos móviles), y esbozar los personajes y otros elementos son algunos de los muchos aspectos que debemos considerar.

Finalmente, una vez que todo esté bien definido, se inicia la fase de programación.

Motores 3D

Al igual que existen los videojuegos en 2D, que utilizan solo dos dimensiones, existen los videojuegos en 3D, los cuales utilizan las tres dimensiones: alto, ancho y profundidad. En estos, podemos movernos en libertad al poder utilizar los tres espacios.

Este tipo de videojuegos utiliza una operación llamada **renderizado**, que consiste en el procesamiento de imágenes en 3D para poder **transformarlas a imágenes 2D**, y así poder **adaptarlas al ojo humano**.

Por norma general, los **objetos 3D** que se usan en los videojuegos están **formados mediante vértices**, formando una serie de **polígonos** que darán lugar al objeto 3D. A mayor cantidad de vértices, tendremos una **mayor cantidad de polígonos**, haciendo que la imagen o modelo 3D sea mucho más realista, pero con el inconveniente de que será **muchísimo más costosa** de procesar.

Una vez que tenemos creados los polígonos, podremos **aplicarles texturas**, las cuales harán que el modelo 3D tenga una imagen mucho más realista.

Los **motores 3D** serán los **encargados de todo esto**, simplificando de una manera bastante considerable todo el proceso de desarrollo del videojuego.

Básicamente, el trabajo del motor 3D será el de:

- crear los modelos,
- moverlos,
- rotarlos según precisemos,
- escalarlos para hacerlos más grandes o más pequeños,
- y un largo etcétera.

Una vez que tenemos todo esto, le podremos **agregar focos de luces** y una **cámara**, lo que dará lugar a una **escena**, que es el **propio videojuego en 3D**.

Este tipo de videojuego **no va a utilizar sprites**, como ocurría en los videojuegos sino que necesitaremos de un programa especial de modelado en 3D para poder crear los personajes que utilizaremos en estos videojuegos. Un ejemplo de programa de modelado en 3D es Blender.

Motores de físicas, IA y sonido

Los movimientos de los personajes en los videojuegos deben ser lo más reales posibles, puede haber caídas, choques, saltos, peleas, etc. Todo esto tiene detrás una gran cantidad de **cálculos matemáticos y físicos**, siendo el **motor de físicas**, que está integrado en los motores (ya sean 2D o 3D), el encargado de realizar todos esos cálculos.

Existen otros motores, como el de inteligencia artificial, que es el encargado de llevar la inteligencia artificial de todos los **personajes** del juego, y el de **sonido**, que será el encargado de **gestionar todos los sonidos del juego**, reproducirlos, pararlos, cambiarlos, etc.

La industria de los videojuegos atrae a numerosas personas no solo por la experiencia de jugar y disfrutar los juegos, sino también por la oportunidad de participar en su desarrollo. Aunque es evidente que contar con una sólida base en programación es esencial para ingresar a este mundo, esto por sí solo no es suficiente, ya que la creación de un videojuego puede evolucionar en un proyecto sumamente ambicioso.

Los equipos de desarrollo de videojuegos se conforman con profesionales especializados en diversas **áreas**, entre ellas:

- Ingeniería de **software**,
- Diseño **gráfico**,
- **Matemáticas** y **física** (dado que los videojuegos involucran una cantidad significativa de cálculos matemáticos y físicos en su desarrollo).
- Además, es crucial contar con un equipo de **marketing** que se encargue del lanzamiento, la comercialización y las ventas del juego.
- Por supuesto, no se puede pasar por alto la importancia de tener equipos de **programadores** especializados en varios lenguajes de programación.

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Desarrollo de videojuegos

2D

3D

Desarrollo de un juego 2D. Interfaz y motor

El primer paso que vamos a llevar a cabo para el desarrollo de un juego en 2D es el diseño de una **interfaz que nos permita jugar** y de un **motor muy básico** que permita a nuestro videojuego funcionar.

Para crear la pantalla, crearemos una clase que herede de **SurfaceView**, que es una clase que nos permitirá **pintar elementos en su interior**, como si se tratase de un lienzo. Al crear esta clase, podremos crear un **objeto** de ella en la **interfaz gráfica**. Esta clase deberá sobrescribir el método `onDraw`, el cual se encargará de dibujar todos los **elementos en la pantalla**, y al que le llegará por parámetro un objeto de la clase **Canvas**, que será nuestro "lienzo".

La clase `PantallaVideojuego` la hemos creado en un paquete llamado `vista`, por lo que para crear un elemento gráfico de este tipo, deberemos escribir el código mostrado en la siguiente figura.

Creando la pantalla en la interfaz gráfica

```
<vista.PantallaVideojuego
    android:id="@+id/pantallajuego"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginLeft="10dip"
    android:layout_marginTop="10dip"
    android:layout_marginRight="10dip"
    android:layout_marginBottom="10dip">
</vista.PantallaVideojuego>
```

Una vez hecho esto, podremos crear una clase que representará el motor de nuestro videojuego.

Para ello crearemos una **clase** que **heredará de Thread** y que tendrá los siguientes **atributos**:

- **FPS**: Será la velocidad, es decir, los **Frames Por Segundo**, a la que se moverá nuestro videojuego.
- Un **objeto** de la clase `PantallaVideojuego`.
- **Running**: Esta variable nos dirá si el videojuego está corriendo o no.
- **Lienzo**: Esta variable del tipo **Canvas** será la encargada de **dibujar los elementos** en pantalla.
- Variables para controlar las pulsaciones en los botones y en pantalla. Controlarán que haya un intervalo de tiempo entre pulsación y pulsación.

Todos los elementos necesarios para la interfaz los podemos encontrar en la parte de recursos del tema, concretamente para este caso, en la versión 1 de las clases:

- PantallaVideojuego.java
- Motor.java

```
public class Motor extends Thread
{
    // Atributos de la clase
    static final long FPS = 25;           // Frames Por Segundo para pintar en el juego
    private PantallaVideojuego pantalla;  // Pantalla donde se va a jugar
    private boolean running = Boolean.FALSE; // Indica si el motor está en marcha o no
    private Canvas lienzo;                // Lienzo para dibujar
    private long ticksPS, startTime, sleepTime; // Variables para actualizaciones
}
```

[...]

```
public class PantallaVideojuego extends SurfaceView
{
    private Bitmap imagenspritebueno, imagenspritemalo, fondo;
    private SurfaceHolder holder;
    private Motor motor;
    private Sprite spritebueno, spritemalo;
    private BitmapDrawable fondorepetido;
}
```

[...]

Fases del desarrollo de videojuegos

En el proceso de desarrollo de videojuegos hay que seguir una serie de fases muy similares a la del desarrollo de cualquier proyecto software, concretamente fase:

- de concreción,
- de diseño,
- de planificación,
- de producción,
- de pruebas,
- de distribución
- y de mantenimiento.

Cada una de ellas es fundamental para un buen resultado de cualquier videojuego.

En esta unidad, iremos un poco **al grano** y no vamos a poder realizar todas y cada una de estas fases, sino que implementaremos un juego básico en 2D paso a paso para que la **parte de producción** que sería la más técnica, quede lo suficientemente clara.

No obstante, no hay que olvidar que el resto de fases poseen igual, o casi mayor importancia...

Agregando los sprites

De unidades anteriores, ya sabemos que los sprites son las imágenes que contienen los movimientos de nuestros personajes. Nosotros vamos a utilizar un sprite bastante simple en el que tendremos movimientos hacia la derecha, izquierda, arriba y abajo, únicamente. Todavía, nada de ataques, muerte, etc.

Para crear sprites personalizados más completos que el que vamos a utilizar, podemos utilizar la siguiente web:

<http://gaurav.munjal.us/Universal-LPC-Spritesheet-Character-Generator/>

Para la **creación de los sprites**, vamos a **necesitar dos elementos**:

- Un enumerado llamado **Direccion** que tendrá las posibles direcciones en las que se moverá nuestro personaje.
- Una clase **Sprite** que será la encargada de la gestión de nuestros sprites.

Con los siguientes métodos:

- Un método para generar una **posición aleatoria** dentro de nuestra pantalla.
- Un método para **cambiar las direcciones** de movimientos de nuestro personaje. Uno por posible dirección.
- Un método para **dibujar nuestro personaje**.
- Un método para **cargar el sprite** del personaje.

Lo primero será agregar en la clase PantallaVideojuego los atributos necesarios para la **gestión de los sprites y el motor**.

A continuación agregamos la **funcionalidad** al motor para que pinte nuestro sprite.

Todos los **elementos necesarios** para el agregado de sprites los podemos encontrar en la parte de recursos del tema, concretamente para este caso, en las clases:

- PantallaVideojuego.java, versión 2

```
public class PantallaVideojuego extends SurfaceView
{
    private Bitmap imagenspritebueno, imagenspritemalo, fondo;
    private SurfaceHolder holder;
    private Motor motor;
    private Sprite spritebueno, spritemalo;
    private BitmapDrawable fondorepetido;
```


- Dirección.java

```
public enum Direccion
{
    ARRIBA("ARRIBA", 3),
    ABAJO("ABAJO", 0),
    IZQUIERDA("IZQUIERDA", 1),
    DERECHA("DERECHA", 2);
    private String stringValue;
    private int intValue;
```

- Sprite.javavideojuego.

```
public class Sprite
{
    /*
     * Configuración del sprite
     *
     * He supuesto las direcciones, casillas del vector
     *
     * dirección = 0 up, 1 left, 2 down, 3 right
     * animación = 3 up, 1 left, 0 down, 2 right
     *
     * En la posición 0 del vector -> imágenes para moverse hacia abajo
     * En la posición 1 del vector -> imágenes para moverse hacia la izquierda
     * En la posición 2 del vector -> imágenes para moverse hacia la derecha
     * En la posición 3 del vector -> imágenes para moverse hacia arriba
     */
    int[] DIRECCION_ANIMACION = { 3, 1, 0, 2 };

    private static final int BMP_ROWS = 4;
    private static final int BMP_COLUMNS = 3;
    private Direccion direccion;
    private int x = 0;
    private int y = 0;
    private static int INCREMENTO_VELOCIDAD = 5;
    private int velocidadEjeX = INCREMENTO_VELOCIDAD;
    private int velocidadEjeY = INCREMENTO_VELOCIDAD;
    private PantallaVideojuego pantalla;
    private Bitmap imagensprite;
    private int frameactual = 0;
    private int ancho;
    private int alto;
    private Random aleatorio;
```

Estructura de los sprites

Una de las cosas que más sorprenden la primera vez que vemos cómo está diseñado un videojuego en 2D es precisamente la configuración de los personajes en los sprites. Lo más lógico es pensar que cada posible movimiento del personaje se guardase en una imagen diferente y que según el movimiento que queramos hacer, cargar una imagen u otra. Esto que parece de sentido común, no es nada eficiente, ya que tendríamos que cargar en memoria una gran cantidad de imágenes simultáneamente en cuanto se iniciara el juego, cosa que, teniendo una única imagen con todos los movimientos, no se tendrá que llevar a cabo de esa forma.

El tener un único sprite tiene también sus inconvenientes, ya que tendremos que saber de antemano dónde están cada uno de los movimientos del personaje dentro del sprite, por ejemplo, el movimiento hacia adelante son los elementos que están en las posiciones (1, 1), (1, 2) y (1, 3), y tendremos que poder obtener esos “fragmentos” de alguna forma.

Empleando siempre la **misma configuración del sprite** en cuanto a **movimientos**, y una vez que podamos “**recortar**” **esos fragmentos en memoria**, el juego será mucho más **rápido y fluido**.

Moviendo al personaje

El siguiente paso va a ser mover a nuestro personaje. En primer lugar, deberemos dar funcionalidad a nuestro motor para poder hacer que se pinte en pantalla el personaje. Esto lo haremos en la función `run`. Básicamente, lo que tendremos que hacer será **bloquear el Canvas** de la pantalla para que solo pueda ser utilizado por el motor, y así indicar a la pantalla que **pinte nuestro personaje**. Así, en cuanto creemos un objeto de nuestra pantalla en la clase principal y lo liguemos con su elemento de interfaz gráfica, tendremos un personaje moviéndose en una única dirección.

En este momento del desarrollo, existirá una estela dejada por el personaje al moverse. Se solventará más adelante.

A continuación, el objetivo es hacer que el personaje se mueva a los lados, y para ello, tendremos que dar funcionalidad a los botones y hacer que llamen a los métodos de cambiar dirección de la pantalla que creamos anteriormente.

El botón de 'pausar' llamará al método `pausar` de la **pantalla**, que a su vez llamará al método de pausar del **motor** del videojuego.

El botón de 'reseteo' servirá para cambiar el sprite de nuestro personaje en caso de tener más de uno.

Código para el movimiento del personaje en `MainActivity.java`

```
@Override
public void onClick(View view)
{
    switch (view.getId())
    {
        case R.id.ibAbajo:
            pantallajuego.cambiarDireccionAbajo();
            break;
        case R.id.ibArriba:
            pantallajuego.cambiarDireccionArriba();
            break;
        case R.id.ibDerecha:
            pantallajuego.cambiarDireccionDerecha();
            break;
        case R.id.ibIzquierda:
            pantallajuego.cambiarDireccionIzquierda();
            break;
        case R.id.ibPausa:
            pantallajuego.pausar();
            break;
    }
}
```

Puedes encontrar la implementación del movimiento al completo en el apartado de recursos del tema, concretamente en el archivo `Sprite.java` versión 2:

```

/**
 * Cambia la dirección de movimiento del sprite hacia la derecha
 */
public void cambiarDireccionDerecha()
{
    if( !movimientoXDerecha() ) // Si el sprite no se está moviendo hacia la derecha
        cambiarSentidoEjeX(); // Cambio el sentido hacia la derecha

    if( direccion == Direccion.ARRIBA || direccion == Direccion.ABAJO )
        // Si el sprite se está moviendo hacia arriba o hacia abajo
    {
        velocidadEjeX = INCREMENTO_VELOCIDAD;
        // Hago que se mueva en el eje x, hacia la derecha o izquierda
        velocidadEjeY = 0;
        // Deja de moverse en el eje y, hacia arriba o abajo
    }

    direccion = Direccion.DERECHA;
    // Hago que el sprite se mueva hacia la derecha
}

```

[...]

Agregado de fondo

El fondo que se utiliza en un videojuego no es más que una imagen donde están dibujados todos los personajes del videojuego.

Podemos tener dos tipos de fondos:

- Fondos de **color**: El fondo del videojuego será un color sólido.
- Fondos de **imagen**: El fondo del videojuego será una imagen que nosotros preparemos previamente.

Para dibujar el fondo del videojuego, basta con hacerlo en la **clase PantallaVideojuego** en el **método onDraw** justo antes de dibujar los personajes, ya que si primero dibujamos los personajes y luego el fondo, este se dibujará encima de ellos y no se verán.

```
public void onDraw(Canvas canvas)
{
    if(canvas != null)
    {
        // Dibujo el fondo del juego
        if(fondorepetido != null)
            fondorepetido.draw(canvas);
        // Dibujo el personaje
        if(spritebueno != null)
            spritebueno.onDraw(canvas);
    }
}
```

Nosotros vamos a utilizar la imagen que se muestra en la siguiente figura como fondo.



Esta imagen **no va a cubrir la pantalla completa de nuestro dispositivo**, así que lo que tendremos que hacer es **dibujarla tantas veces como sea necesario** para que cubra la totalidad de la pantalla, pero ¿cómo podemos hacer esto?

Utilizaremos para ello las clases `Bitmap` y `BitmapDrawable`, y sus **métodos** `setBounds`, `setTileModeX` y `setTileModeY` para hacer que **repita la imagen** hasta rellenar.

Puedes encontrar la implementación del fondo al completo en el apartado de recursos del tema, concretamente en el archivo `Pantallavideojuego.java` versión 3.

Motores comerciales y open source

Como ya vimos en la unidad anterior, los motores de los videojuegos son entornos de desarrollo especialmente enfocados a facilitar la programación de este tipo de software. Como casi todo software, los motores pueden ser privativos u open source.

Ejemplos de motores de videojuegos **privativos** pueden ser

- Unity 3D,
- Unreal Engine.

Ejemplos de motores de videojuegos **open source** pueden ser

- Xenko,
- Godot Engine.

Algunas de las propiedades con las que trabajan los motores son **luz, texturas, reflejos, sombras...** que permitirán hacer al videojuego lo más **realístico** posible. Con respecto a las sombras, se utilizan lo que se conoce como **Shaders**, que son los elementos encargados de gestionar todo lo relacionado con las sombras de los objetos y personajes del juego, contribuyendo así, a otorgar el **mayor realismo** posible.

Agregando los enemigos

Los enemigos serán otro personaje cargado con un **sprite** diferente, que se estará moviendo de forma aleatoria en nuestra pantalla. El sprite de nuestro enemigo será el mostrado en la siguiente figura.



Como podemos observar, es muy parecido al sprite de nuestro personaje, de hecho, podríamos intercambiarlos y no provocaríamos ningún error en la implementación a nivel de código. Para el agregado del enemigo, deberemos crear otro **objeto de tipo sprite** en nuestra clase **PantallaVideojuego** y después seguir los siguientes pasos:

```
private Sprite spritemalo;
```

- En el método `onDraw`, deberemos pintar al enemigo después de pintar el fondo de pantalla.

```
if(spritemalo != null)
    spritemalo.onDraw(canvas);
```

- En el método `funcionalidad`, deberemos cargar y crear el sprite de nuestro enemigo.

```
spritemalo = new Sprite(this, imagenspritemalo);
```

- Una vez hecho esto, podremos compilar nuestro videojuego y tendremos el resultado que se muestra en la siguiente figura.



Una posible mejora podría ser la incorporación de sonido al juego, por ejemplo, al aplastar a un enemigo. Para ello, podemos emplear las clases MediaPlayer y SoundPool.

Puedes encontrar la implementación de la inclusión del enemigo al completo en el apartado de recursos del tema, concretamente en el archivo Pantallavideojuego.java versión 4.

Detectando las colisiones

Exploraremos cómo detectar colisiones entre dos sprites en un videojuego 2D. Recordemos que una colisión ocurre cuando dos sprites se cruzan o, en otras palabras, cuando ocupan una cierta parte de la pantalla simultáneamente. Para lograr esto, hemos creado una función llamada "hayColision" en nuestra clase "Sprite".

```
public boolean hayColision(float x2, float y2)
{
    return (x2 > x) && (x2 < (x + ancho)) && (y2 > y) && (y2 < (y + alto));
}
```

Esta función toma como parámetros las **coordenadas del sprite** con el que queremos verificar la colisión y devuelve verdadero o falso, indicando si se ha producido la colisión o no.

La función realiza el cálculo dentro de este código para determinar si las coordenadas (x2, y2) que le proporcionamos están dentro del área cubierta por las coordenadas del sprite con el que realizamos la llamada.

Además, hemos creado dos métodos adicionales, "getX" y "getY", que devuelven las posiciones X e Y del sprite, es decir, su ubicación en el momento de la llamada.

```
/**
 * Devuelve la coordenada X del sprite
 * @return Posicion X del sprite
 */
public float getX()
{
    return x;
}

/**
 * Devuelve la coordenada x del sprite
 * @return Posición y del sprite
 */
public float getY()
{
    return y;
}
```

Nos dirigimos a la clase "PantallaVideojuego". En el método "onDraw", donde dibujamos elementos en pantalla, incluiremos el fondo y los sprites de nuestro personaje y del enemigo. Luego, verificaremos si hay una colisión entre el sprite de nuestro personaje y las coordenadas X e Y del enemigo. Si se detecta una colisión, se imprimirá un mensaje en la consola indicando "Colisión detectada".

```
if( spritebueno.hayColision(spritebueno.getX(), spritebueno.getY()) )
{
    System.out.println("COLISION DETECTADA");
}
```

Finalmente, ejecutamos nuestra aplicación para observar el resultado. En la consola, al hacer que nuestro personaje colisione con el enemigo, verificamos que se ha detectado una colisión entre los dos personajes.

Agregando texto al juego

Aprenderemos cómo dibujar texto en la pantalla de nuestro videojuego 2D. Para lograrlo, nos dirigimos a la clase "PantallaVideojuego" y creamos una función llamada "dibujarTexto", la cual se encargará de dibujar el texto en el lienzo. Esta función aceptará los siguientes parámetros:

- "texto" (el texto a dibujar),
- "x" e "y" (las coordenadas donde se ubicará el texto dibujado),
- "tamañoTexto" (el tamaño del texto),
- "rojo", "verde" y "azul" (componentes RGB del color del texto),
- y "canvas" (el lienzo donde se dibujará el texto).

Utilizaremos un objeto del tipo "Paint" para dibujar el texto en la pantalla.

Crearemos una fuente del tipo "Typeface", en este caso, será de tipo "Serif", cursiva y en negrita. Asignamos esta fuente a nuestro pincel para que dibuje con ella.

Utilizaremos el método "drawText" de "canvas" para pintar el texto deseado. El color se especificará en formato ARGB, por lo que se deberá proporcionar un grado de transparencia. En este caso, se establece en 255, lo que significa que no será transparente; si fuese 0, sería totalmente transparente.

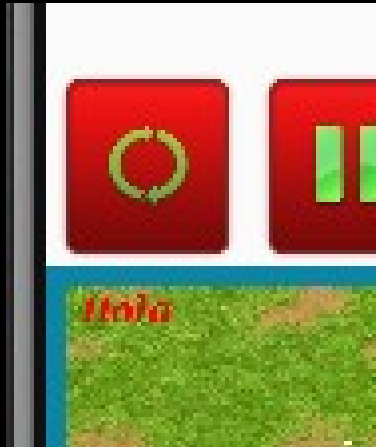
```
private void dibujarTexto(String texto, int x, int y, int tamatexto, int rojo, int verde, int azul,
Canvas canvas)
{
    // no será transparente:
    int transparencia = 255;
    Paint pincel1 = new Paint();
    /*
     * Configuramos el color del texto, que ha de estar en formato ARGB
     */
    pincel1.setARGB(transparencia, rojo, verde, azul);
    pincel1.setTextSize(tamatexto);
    pincel1.setTypeface(Typeface.SERIF);
    @SuppressWarnings("WrongConstant") Typeface tf = Typeface.create(Typeface.SERIF,
Typeface.ITALIC | Typeface.BOLD);
    pincel1.setTypeface(tf);
    canvas.drawText(texto, x, y, pincel1);
}
```

Después de dibujar todos los elementos del juego y verificar colisiones en el método "onDraw", añadiremos la instrucción para dibujar el texto "hola" en la posición 20-50, con un tamaño de texto de 50 y utilizando los componentes de color proporcionados, lo cual resultará en un color rojo.

```
// Dibujo el texto  
dibujarTexto("Hola", 20, 50, 50, 255, 0, 0, canvas);
```

Al ejecutar el juego, observaremos el resultado:

el texto "hola" en rojo, en cursiva y en negrita en la pantalla.



Concepto de animación en el desarrollo de videojuegos 3D

El desarrollo de videojuegos en 3D es algo más complejo de realizar que el de 2D, debido a que interviene una dimensión más: la profundidad, lo que complicará los movimientos de los personajes de forma exponencial. Este tipo de videojuego **no se presta tanto a la realización desde cero, como** hemos hecho en los puntos anteriores con el **juego en 2D**.

En un videojuego 3D, ¿cómo se representan los personajes? Mediante modelos.

A continuación, vamos a conocer una serie de **conceptos básicos** para la creación de un videojuego en 3D:

- **Mundo:** El concepto de mundo en los videojuegos 3D se refiere al **espacio** donde se desarrollará el videojuego, es decir, por donde podrán **moverse los personajes** del mismo.
- **Sistema de coordenadas:** Los videojuegos en 2D tenían únicamente dos coordenadas, el alto y el ancho, sin embargo, cuando tratamos con videojuegos en 3D, tendremos las tres dimensiones del espacio, que son alto, ancho y profundidad. Este sistema de coordenadas tendrá que ser representado de una forma totalmente diferente al de dos dimensiones, utilizando ejes cartesianos con **tres coordenadas: X, Y, y Z**.
- **Vectores:** Un vector no es más que un **conjunto de coordenadas, tres** en el caso de los videojuegos 3D, que representarán el **punto exacto del mundo** donde se encuentra un **objeto**.
- **Creación de componentes:** Para la creación de los componentes del mundo en los videojuegos 3D, no se utilizarán sprites, sino que tendremos que **crear figuras** mediante el uso de:
 - **vértices,**
 - **polígonos,**
 - **mallas** de polígonos.
- **Grafo de escena:** Los grafos de escena van a contener toda la **estructura de nuestro videojuego 3D en su conjunto**. Tienen **aspecto de árbol** y de él “colgarán” todos los elementos del mismo. Es un **análogo** a cuando **pintamos los elementos** en el motor de nuestro videojuego **2D**.

Grafos de Escena en Videojuegos: Estructura y Funciones

Exploraremos qué son los grafos de escena en un videojuego y cuál es su utilidad. Un grafo de escena es un **grafo dirigido acíclico** compuesto por **nodos** que contiene los datos que definen un **escenario virtual** y controlan su **proceso de dibujado**.

Su **función principal** es almacenar la información del escenario virtual, representando todos los **objetos** del videojuego **mediante nodos**.

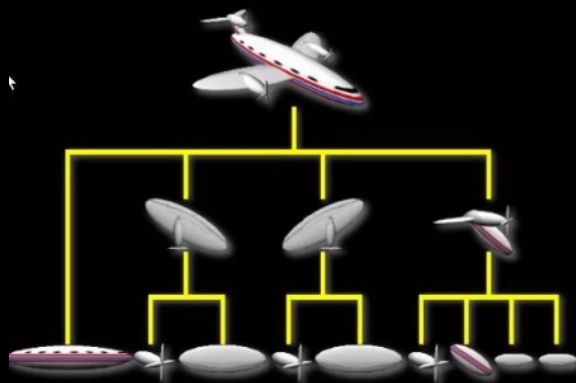
Existen diversos **tipos de nodos**, algunos de los cuales almacenan **información geométrica** de los objetos en la pantalla.

Las **funciones clave de un grafo de escena** en un videojuego son:

1. **Organización Lógica de la Escena:** Contribuye a establecer una estructura ordenada y conexa en la escena virtual.
2. **Dependencias Jerárquicas:** Establece dependencias jerárquicas entre los nodos de cada objeto, facilitando la organización.
3. **Selección de Nodos:** Permite la selección de varios nodos, incluso en diferentes niveles.
4. **Proceso de Culling (sacrificio):** Posibilita la **eliminación automática** de objetos fuera del campo de visión.
5. **Acceso Cómodo a Librerías Gráficas:** Facilita el acceso a librerías gráficas de bajo nivel, como **OpenGL**.

En este ejemplo de un **grafo de escena** de un avión, observamos **tres niveles**:

1. el superior con el **avión completo**.
2. dos niveles más donde se **descompondrán las piezas** del avión, como el cuerpo del avión que se muestra.



Un grafo de escena es fundamental para establecer una **organización lógica y jerárquica** en los videojuegos, permitiendo un **control eficiente** de los objetos y mejorando la interacción visual.

Escena de un juego real

Como ya sabemos después del estudio del tema, un videojuego se va a componer de multitud de motores y elementos.



Un ejemplo claro del tipo de componentes que podemos encontrar en un juego, lo tenemos en la imagen mostrada , que representa una escena real del videojuego *Dark Souls Remastered*, en la que podemos apreciar varios elementos como:

- nuestro **personaje** de espaldas,
- un **dragón** al fondo,
- y el escenario.

Dentro del escenario podemos observar un **juego de luz** que incide desde la parte trasera, haciendo que resulten sombras, como por ejemplo la de nuestro personaje.

También se puede apreciar la **interfaz del juego**, que se compone de

- la **vida y estamina**, en la parte superior,
- de nuestra **arma, escudo y número de almas**, en la parte inferior.

A groso modo, podemos deducir que como mínimo se están utilizado **tres grafos**:

- uno del personaje,
- otro del dragón
- y otro del propio escenario.

Te invitamos a que sigas analizando la imagen para que descubras más grafos y elementos...

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Introducción al lenguaje Swift

Variables

Constantes

Operadores

Arrays

Introducción a Swift

Vamos a aprender las bases del lenguaje de programación usado para el desarrollo de aplicaciones móviles para dispositivos con **sistema operativo iOS: Swift**.

Ya sabemos que para desarrollar aplicaciones para dispositivos Android, se pueden utilizar, entre otros, dos lenguajes: Java y Kotlin. Algo similar ocurre para el desarrollo de aplicaciones para **dispositivos iOS**. En estos, podremos utilizar los lenguajes [Objective-C](#) y [Swift](#). **En un principio**, las aplicaciones se desarrollaban íntegramente en [C#](#), pero **hoy en día**, se están migrando de una forma bastante rápida a **Swift**.

El lenguaje de programación Swift fue desarrollado por Apple y es:

- un lenguaje **multiparadigma**,
- soportando:
 - **orientación a objetos**,
 - **orientación a protocolos**,
 - **programación funcional**
 - y **programación imperativa**.

Mediante este lenguaje se podrán realizar aplicaciones tanto para dispositivos **móviles con iOS** como aplicaciones para el **sistema operativo macOS**.

Apple presentó este nuevo lenguaje de programación en el año **2014**, en la [WWDC \(Worldwide Developers Conference\)](#), que es como se denominan las **conferencias ofrecidas por Apple** para **presentar sus productos**.

A pesar de ser un lenguaje de programación diseñado por Apple, lo presentaron como **software libre**, algo bastante contrario a la hermeticidad de esta compañía. Además, este lenguaje es multiplataforma, con lo cual, podremos **desarrollar** aplicaciones escritas **en él** en cualquiera de los tres grandes sistemas operativos: **GNU/Linux, Windows y, por supuesto, macOS**.

Instalación de Swift en Windows

Para instalar Swift en Windows, necesitamos descargar el programa **Swift for Windows**, el cual nos instalará el **compilador de Swift** para Windows y un programa para poder compilar y ejecutar los programas que desarrollemos.

Para descargarlo, nos haremos de la siguiente página:

<https://swiftforwindows.github.io/>

Seleccionamos Download y empezará la descarga.

Una vez descargado, lo instalaremos pulsando siguiente en todas las pantallas del asistente.

Una vez instalado, tendremos el **compilador listo**. Los programas que estén escritos en Swift deberán tener la **extensión .swift**. Desde Swift 2.0 podremos **compilar un programa** pulsando el botón Compile y seleccionando el fichero de código, y para poder **ejecutar en una terminal** el programa, pulsaremos el botón **Run** y seleccionaremos el fichero.

Para evitar llevar a cabo estos pasos, vamos a utilizar el programa Visual Studio Code, que integra una terminal en el propio editor y nos permitirá escribir el código, compilarlo y ejecutarlo de forma muy sencilla.

Recordamos que para descargar Visual Studio Code, lo podemos hacer desde su página web:

<https://code.visualstudio.com/download>

Una vez instalado y con un **fichero de código abierto**, podremos pulsar en el menú Terminal New Terminal y nos aparecerá una terminal integrada en la que podremos compilar nuestros programas, accediendo primero a la ubicación del compilador con el comando:

```
cd C:\Swift\bin
```

y, posteriormente:

- Para compilar: `.\swiftc.exe ruta_fichero.swift`
- Para ejecutar: `.\swift.exe ruta_fichero.swift`

Instalación de Swift en otros sistemas operativos

Para instalar el compilador de Swift en GNU/Linux tendremos que descargarlo de la siguiente página web <https://swift.org/download> y seleccionar la distribución deseada según nuestro sistema. Una vez descargado lo descomprimos y lo copiamos en la carpeta `/usr/`, combinando todos los ficheros que nos indique.

En los sistemas operativos de MacOS ya viene instalado por defecto.

Dónde desarrollar con Swift

El lenguaje de programación Swift es un lenguaje multiplataforma, que, si recordamos, quiere decir que se podrá instalar en los tres grandes sistemas operativos del mercado: GNU/Linux, Windows y macOS.

Visto esto, podremos desarrollar programas escritos en Swift en cualquiera de estos sistemas operativos, pero esto no implica que se puedan desarrollar aplicaciones para iOS desde ellos. **El desarrollo de aplicaciones para iOS se podrá realizar única y exclusivamente desde el sistema operativo macOS.** Esto quiere decir que desde GNU/Linux y Windows podremos realizar programas escritos en Swift sin ningún inconveniente, pero **no podremos realizar aplicaciones para móviles iOS.**

Algo totalmente distinto ocurre con Android, ya que Android Studio se puede instalar en cualquiera de los tres grandes sistemas operativos.

Comentarios al código

Todos los lenguajes de programación deben de proporcionar un método para la implementación de comentarios en el código.

Ya sabemos que los comentarios nos ayudarán a introducir aclaraciones en el código, que nos facilitarán entender cómo están implementados los algoritmos que utilizemos para resolver cada problema en cuestión.

En los comentarios, recordemos que podremos escribir lo que necesitamos, incluso haciendo uso de cualquier símbolo o carácter que no podamos utilizar a la hora de programar, como pueden ser las tildes, la letra ñ, etc.

Swift no es una excepción y también proporciona un mecanismo de comentarios bastante simple y fácil de utilizar.

En primer lugar, tenemos los **comentarios de una única línea**, que nos ayudarán a escribir comentarios aclaratorios de una línea de longitud. Estos comentarios se implementan mediante **dos barras inclinadas**, y todo lo que se sitúe a partir de ellas, será considerado como comentario al código que le preceden.

```
// Este es un comentario de una sola línea con ñ y öôô
```

El otro tipo de comentario que nos permite implementar Swift es el comentario multilínea, en el cual, podremos escribir todas las líneas que necesitemos, considerándose como un único comentario. Estos comentarios podrán **escribirse entre /* y */**.

Comentario multilínea

```
/*  
Esto es un ejemplo de  
comentario de  
varias líneas  
*/
```

Como podemos observar, la implementación de los comentarios que nos proporciona Swift es idéntica a la que nos proporciona Java, por lo que no vamos a tener ninguna dificultad en adaptarnos a ella.

El color verde que tienen los comentarios en estos ejemplos es debido al uso de Visual Studio Code.

Variables y operaciones

El lenguaje de programación Swift nos ofrece un gran abanico de variables que podremos utilizar en nuestros programas, pero, al fin y al cabo, los tipos más complejos de datos que podremos utilizar van a estar basados en los tipos de datos básicos, como ocurría en los demás lenguajes de programación.

Los tipos de datos básicos que vamos a poder utilizar en Swift son los siguientes:

- **Int**: Este tipo de dato representa los números enteros con **32 bits** con un rango de entre menos 2.147.483.648 y mas 2.147.483.647.
- **Double**: Este tipo de datos representa los números reales con **64 bits** con hasta **15 decimales** de precisión.
- **Bool**: Este tipo de datos representa los datos booleanos que puede ser **0 (false)** o 1 (true).
- **String**: Este tipo de datos representa una cadena de caracteres.

En Swift, las variables, **al declararse, no llevarán indicado un tipo**, obteniéndolo en el momento en el que se igualan a algo por primera vez. Para declarar una **variable**, utilizaremos la palabra reservada **var**, mientras que para declarar una **constante**, utilizaremos la palabra reservada **let**. También **será posible indicar el tipo al declarar las variables**, aunque esto sería un poco inusual.

Ejemplo de declaración de variables en Swift

```
// Declaración de variables enteras
var numero1 = 8, numero2 = 3
// Declaración de variables de un tipo concreto
var palabra:String
palabra = "Hola"
// Declaración de una constante real
let PI = 3.141592
```

Las **operaciones** que podremos hacer con variables son las mismas que ya conocemos:

- Para datos del **tipo numérico**: Suma, resta, producto, división y **cálculo del resto**.
- Para datos del tipo **cadena**: **Concatenación** y demás operaciones que permita la clase.

Las variables en Swift **siempre deberán tener un valor**, es decir, no se podrán tener variables sin inicializar.

Variables opcionales

Swift no admite variables sin inicializar, sino que **se establecerá en el futuro**, pero **sí** que podemos **inicializar una variable sin saber qué valor** tendrá en ese momento,.

Esto se hace con un **'?' tras el tipo** de variable.

Con la interrogación, estamos indicando a Swift que la variable es de un tipo, pero que aún no queremos o no podemos asignarle un valor, por lo que se le **asignará automáticamente un valor nulo**.

Cuando queramos **recuperar dicho valor** tendremos que añadir el **modificador '!' (fin de exclamación)** para que extraiga de este contenedor opcional el valor, y lo **muestre tal cual**.

Entrada y salida de datos

Cualquier lenguaje de programación que se precie necesitará proveer algún mecanismo para poder leer información de teclado y mostrar información por pantalla.

Para **leer datos de teclado** en Swift, utilizamos la función `readLine()`, la cual nos **devolverá siempre un String**.

En el caso de que queramos leer un **entero o un real**, tendremos que hacerles un **casting** para convertir el String que hemos leído al dato que queramos.

Esta función nos va a **devolver una variable opcional**, por lo que, para usarla, deberemos **tratarla como tal**, y usar el **operador '!' (fin de exclamación)** cuando la invoquemos.

Para poder imprimir por pantalla en Swift, vamos a utilizar la **instrucción print**, de la forma `print("Hola Mundo")`.

Otro elemento que podremos utilizar son los especificadores para indicar que, en ese lugar, irá el **contenido de una variable (parecido al f "{ }" en python)**. En Swift, podemos hacerlo de la forma `\(variable)` (barra inclinada a izquierda seguido de la variable entre paréntesis). A esto lo conocemos como **interpolación de strings**.

Un aspecto que tenemos de tener en cuenta es que en Swift las **instrucciones no terminarán con un punto y coma (;)**, al igual que ocurría en Java. No obstante, en el caso de que queramos anidar múltiples instrucciones **en una única línea** de código, sí que tendremos que **separarlas mediante punto y coma**, de forma que el compilador las pueda reconocer sin problema.

Lo vemos en el siguiente ejemplo:

```
var numero = 9 ; print("El número es \(numero)")
```

Ejemplo de entrada y salida en Swift

```
// Leo un número entero por teclado
print("Introduce un entero")
var enterostring = readLine()!
var numero = Int(enterostring)!
// Leo un número real por teclado
print("\nIntroduce un número con decimales")
var realstring = readLine()!
var real = Double(realstring)!
// Mostramos los resultados
print("\nEl número entero es: \(numero)")
print("El número real es: \(real)")
```

Ejemplo de Entrada y Salida de Información en Swift

Introducción:

En este ejemplo práctico, abordaremos detalladamente la entrada y salida de información en Swift. El programa que desarrollaremos solicitará al usuario un **número entero**, un **número real** y una cadena, para posteriormente mostrarlos en la pantalla.

La instrucción 'print' será utilizada para imprimir mensajes en la pantalla, incluyendo la capacidad de introducir saltos de línea con '\n' y tabuladores con '\t'.

Código:

```
import Foundation
// Bloque 1: Lectura de un número entero
print("Introduce un entero")
var enterostring = readLine()!
var numero = Int(enterostring)!
// Bloque 2: Lectura de un número real
print("\nIntroduce un número con decimales")
var realstring = readLine()!
var real = Double(realstring)!
// Bloque 3: Lectura de una cadena
print("\nIntroduce una cadena")
var cadena = readLine()!
// Bloque 4: Mostrar resultados
print("\nEl numero entero es: \(numero)")
print("El numero real es: \(real)")
print("La cadena es: \(cadena)")
```

Explicación del Código:

- En el primer bloque, solicitamos y leemos un número entero, mostrando un mensaje al usuario y creando la variable 'enteroString'.
- En el segundo bloque, realizamos un proceso similar para un número real, utilizando casting a 'Double'.
- En el tercer bloque, leemos una cadena y la almacenamos en la variable correspondiente.
- En el cuarto bloque, mostramos los resultados utilizando la función 'print' y presentamos los valores de las variables.

Compilación y Ejecución:

Para compilar y ejecutar este ejercicio, utilizaremos el archivo 'shiftc.exe' ubicado en 'C:\shift\bin', proporcionándole la ruta del archivo como argumento.

Ejemplo de Ejecución:

Ejecutando el programa...

Introduce un número entero: 7

Introduce un número real: 5.3

Introduce una cadena: hola

Resultados:

Número entero: 7

Número real: 5.3

Cadena: hola

Este formato mejorado facilita la lectura y comprensión del código, proporcionando una estructura más clara para su estudio.

Arrays

En el lenguaje de programación Swift, los arrays son un tipo de dato básico.

El concepto de array sigue siendo exactamente el mismo que hemos estudiado en otras asignaturas, es decir, seguiremos pudiendo tener una colección de datos del mismo tipo y realizar operaciones sobre ellos.

Para declarar un array en Swift, tendremos que igualar una variable a una serie de valores colocados **entre corchetes** ([]) y **separados mediante comas**. También podremos declarar arrays que sean **constantes mediante let**.

Un ejemplo de declaración de un array puede ser el siguiente:

```
var array = [1, 2, 3, 4, 5]
```

Al igual que con las variables, **podremos indicar el tipo** de elementos que contendrá el array, aunque **no sea obligatorio**. Podremos hacerlo de la siguiente forma:

```
var palabras: [String] = ["Hola", "que", "tal"]
```

```
let palabrasconstantes: [String] = ["Buenos", "días"]
```

Se podrá **mostrar un array por pantalla directamente con la instrucción print** tratándolo como una **variable** normal y corriente.

Suma de arrays en swift:

```
let primera = ["A","B"]
```

```
let segunda = ["C","D"]
```

```
let tercera = primera + segunda
```

La **operación + en Swift es utilizada para concatenar dos arrays**, y en este caso, tercera contendrá los elementos de ambas arrays concatenados en orden.

Al ejecutar el código Swift:

```
var numeros = [1, 2, 3]
```

```
numeros += [4]
```

El array tendrá 4 elementos:

La operación '+=' se utiliza para agregar elementos a un array en Swift. En este caso, se agrega el número 4 al array 'números', por lo que después de la ejecución, el array contendrá los elementos [1, 2, 3, 4]. Sin embargo el test dice que el código dará error...

En Swift, el operador de incremento clásico **++** **no está disponible**. En su lugar, se recomienda utilizar la forma **+=1** para incrementar una variable en una unidad. El operador **++** ha sido eliminado en Swift, y se fomenta el uso de la sintaxis más explícita.

El lenguaje Swift nos ofrece una gran cantidad de **métodos** para poder operar con **arrays**, siendo algunos de ellos los que podemos ver en la siguiente tabla.

Métodos de los arrays en Swift

Función → Funcionalidad

count → Devuelve la cantidad de elementos del array.

append(valor) → Inserta un valor **al final** del array.

insert(valor, at: posición) → Inserta un valor en el array en la **posición indicada**.

remove(at: posición) → Elimina el elemento de la **posición indicada**.

Sort() → Ordena en orden ascendente el array.

sort(by: >) → Ordena en orden **descendente** el array.

Reversed() → **Invierte** los elementos del **array**.

Shuffle() → **Desordena** de forma aleatoria los elementos. Este método funciona a partir de Swift 4.2.

array[posición] = nuevovalor → Asigna un **nuevo valor** a la posición del array indicada.

print(array) → Imprime por consola el **array completo**.

Ejemplo de arrays

Introducción:

Presentamos a continuación un ejemplo práctico de cómo trabajar con Arrays en Swift. Comenzamos **declarando un Array** de enteros vacío.

Luego, se leen tres datos de tipo entero (número uno, número dos y número tres) que se agregan al Array utilizando el **método 'append'**.

El contenido del Array se muestra mediante 'print' antes y después de ordenarlo, con el objetivo de verificar la efectividad de la operación de ordenamiento.

Proceso:

1. Declaración del Array:

- Se inicia declarando un Array de enteros vacío utilizando la sintaxis

```
var Array: [Int] = []
```

2. Lectura de Datos:

- Se solicitan tres datos de tipo entero (número uno, número dos y número tres) al usuario mediante la función **readLine()**.

3. Agregar Datos al Array:

- Los datos leídos **se agregan** al Array usando el método **append**.

4. Visualización del Array:

- Se muestra el contenido del Array mediante '**print**' antes de realizar el ordenamiento.

5. Ordenamiento del Array:

- El Array se ordena utilizando el método '**sort()**'.

6. Visualización del Array Ordenado:

- Se muestra el **Array después de ordenarlo**, confirmando el éxito del proceso.

Compilación y Ejecución:

Se utiliza el compilador de swift para compilar y ejecutar el fichero del ejercicio. Durante la ejecución, se solicita al usuario ingresar tres números, y luego el programa muestra el Array antes y después de ordenarlo.

```
import Foundation

// Me declaro las variables
var numero = 0
var array: [Int] = []

// Introducimos tres elementos en el array
print("Introduce un entero")
var enterostring1 = readLine()!
var numero1 = Int(enterostring1)!

print("Introduce un entero")
var enterostring2 = readLine()!
var numero2 = Int(enterostring2)!

print("Introduce un entero")
var enterostring3 = readLine()!
var numero3 = Int(enterostring3)!

// Agregamos los elementos al array
array.append(numero1)
array.append(numero2)
array.append(numero3)
```

```
print("El array es \n(array)")

array.sort()

print("El array ordenado es \n(array)")
```

resultado:

Introduce un entero

4

Introduce un entero

2

Introduce un entero

8

El array es [4, 2, 8]

El array ordenado es [2, 4, 8]

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Swift

Condicionales if y switch

Iteración con for

Clases

Herencia

Estructuras condicionales

El lenguaje de programación Swift nos permite utilizar sentencias condicionales tal y como lo ofrece cualquier otro lenguaje.

En primer lugar, vamos a ver cuál es la [sintaxis de la estructura if](#):

Después de la palabra reservada `if`, deberemos colocar una **condición**, que irá **sin paréntesis**, y a continuación, el bloque que se ejecutará en caso de que la condición sea verdadera. Los **bloques** de código se delimitarán mediante **llaves** (`{}`).

De igual forma que en otros lenguajes, aquí también disponemos del **bloque else**, cuyo bloque se ejecutará cuando la condición sea falsa.

Un ejemplo de if-else lo podemos ver en el siguiente código:

Ejemplo de if-else

```
print("Introduce un entero")
var numerostring = readLine()!
var numero = Int(numerostring)!
if numero % 2 == 0
{
    print("El numero es par")
} else { print("El numero es impar")
}
```

Si queremos evaluar varias condiciones anidadas en un mismo if, podremos utilizar los operadores lógicos AND (representado mediante `&&`) y OR (representado mediante `||`).

Swift también nos ofrece la posibilidad de usar la estructura condicional `switch`, a la que tendremos que indicar una variable para evaluar, y realizar acciones mediante sus valores (usando la palabra reservada `case`).

En este caso, tendremos que poner de forma **obligatoria la opción default**, que se ejecutará cuando no se cumplan ninguno de los casos evaluados.

Ejemplo de switch

```
print("Introduce un entero")
var numerostring = readline()!
var numero = Int(numerostring)!

switch numero
{
    case 1:
        // código
    case 2:
        // código
    case 3:
        //código
}
```

Estructuras repetitivas

Como podemos imaginar, cualquier lenguaje de programación que se precie deberá ser capaz de proporcionar estructuras repetitivas, o bucles, para poder realizar tareas iterativas con ellas.

El bucle `for` es el primero de los que vamos a ver en el lenguaje Swift. Este bucle iterará en un intervalo de valores, por ejemplo, de 1 a 10, recorriéndolo de uno en uno.

Para crear un bucle `for`, deberemos escribir la palabra reservada `for` y una `variable iteradora`, seguida de `in` y un intervalo, que se definirá de la forma **inicio, tres puntos, fin**.

Ejemplo de for

```
for i in 1 ... 10
{
    print("Voy por \(i)")
}
```

También podremos `recorrer` fácilmente los `elementos de un array` mediante un **bucle for**:

Ejemplo de for recorriendo un array

```
var array_de_valores = [1, 2, 3, 4, 5, 6]
for valor in array_de_valores
{
    print("Valor vale \(valor)")
}
```

El otro tipo de bucle que nos proporcionará Swift es el bucle `repeat while`. Este bucle se estará **ejecutando mientras** se cumpla una cierta **condición**.

Para crear un bucle `repeat while`, deberemos escribir la palabra `repeat` y un **bloque con código** a ejecutar. Por último, se escribirá la palabra `while` seguida de la **condición** de parada del bucle.

A estas condiciones, podremos aplicarles los operadores AND y OR de igual forma que lo hemos podido hacer con otros lenguajes.

Ejemplo de repeat while

```
var numero:Int?  
  
repeat {  
    print("Introduce un entero")  
    let numerostring = readLine()!  
    numero = Int(numerostring)!  
} while numero! % 2 != 0
```

En el ejemplo de la figura anterior, estaremos leyendo números por teclado hasta que se introduzca un número par, que hará que no se cumpla la condición y se detenga el bucle.

Cláusula for(from - through - by)

Si recordamos los for en Java, a estos podíamos **indicarles** que su **incremento** fuese en el número que nosotros necesitáramos, ¿se podrá hacer lo mismo en Swift? La respuesta es que sí, en este lenguaje, podremos utilizar la cláusula **from - through - by** mediante la que podremos indicar desde **dónde** queremos **empezar** a iterar, **hasta dónde** queremos llegar, y su **intervalo**.

Podría utilizarse de la forma que se indica en el siguiente código.

Ejemplo de from - through - by

```
print("Introduce un entero de inicio")
var numero1string = readLine()!
var numero1 = Int(numero1string)!

print("Introduce un entero de fin")
var numero2string = readLine()!
var numero2 = Int(numero2string)!

for num in stride(from: numero1, through: numero2, by: 2)
{
    print("Recorremos los números, y encontramos el impar: \(num)")
}
```

Métodos

En Swift, podremos declarar funciones o métodos mediante la palabra reservada `func`, seguida del **nombre** que queramos darle al **método**.

Después del nombre, deberemos indicar *entre paréntesis la lista de parámetros* que recibirá la función, habiendo de **indicar el tipo** de los mismos. En el caso de que no se reciban parámetros, se usarán los paréntesis vacíos.

Por último, con el operador flecha (`->`), indicaremos el **tipo** del valor **que devolverá**, siendo `Void` en caso de no devolver nada. El código del mismo irá en un **bloque entre paréntesis**.

Para devolver los valores, utilizaremos `return`.

Ejemplo de declaración de funciones

```
// Función que recibe dos parámetros y no devuelve nada
func funcion1(parametro1:Int, parametro2:String) → Void
{

}

// Función que no recibe parámetros y devuelve un entero
func funcion2() → Int
{
    return 1
}
```

Una vez que hemos declarado las funciones necesarias, deberemos poder **llamarlas** para poder utilizarlas.

- Cuando se trate de funciones que **devuelvan un valor**, deberemos **igualarlas a una variable o constante**,
- mientras que cuando se trate de **funciones que no devuelvan** nada, podremos invocarlas **directamente** escribiendo su nombre.

En Swift, es importante remarcar que deberemos **escribir los nombres de los parámetros** en las **llamadas** de forma obligatoria.

Ejemplo de llamadas a funciones

```
// Llamada a la función con parámetros
var valor = "Hola"
fun1(parametro1: 3, parametro2: valor)

// Llamada a la función que devuelve un valor
var resultado = funcion2()
```

Parámetros con valores por defecto

Vamos a ver un ejemplo de funciones en Swift con parámetros con valores por defecto.

Como podemos ver, hemos creado una función llamada 'unirCadenas' a la que le pasamos dos parámetros: 'separador' del tipo Character y 'cadenas' del tipo Array de String.

Esta función nos devolverá un String. Aquí podemos ver que el parámetro 'separador' está igualado a un espacio en blanco. Esta es la forma correcta de hacer que un [parámetro de una función tenga un valor por defecto](#). A partir de ahora, en el caso de que llamemos a la función 'unirCadenas' y no le pasemos el parámetro 'separador', su valor será automáticamente espacio en blanco.

Esta función lo que hace es que va a unir todos los String del Array 'cadenas' separándolos mediante el elemento 'separador', creando una nueva cadena y devolviéndola.

Aquí podemos ver que hemos llamado a nuestra función dos veces: una pasándole un separador que es el guión y otra sin pasarle ningún separador. Después mostramos el resultado de ambas llamadas.

Vamos a compilar nuestro ejemplo para ver si no tiene ningún error y lo ejecutaremos.

Se ha pasado la compilación correctamente y ahora lo ejecutamos.

```
El resultado 1 vale: valor1-valor2-valor3-
```

```
El resultado 2 vale: valor1 valor2 valor3
```

Aquí podemos ver que tenemos dos resultados:

los elementos valor1, valor2 y valor3 que le hemos pasado a la función, pero en el primer caso estarán separados mediante un guión y en el segundo mediante un espacio en blanco, ya que no hemos pasado el parámetro 'separador'.

Código

```
import Foundation

/
 * Esta función une varias cadenas con un separador indicado
 * @param separador Este es el separador para separar las cadenas. Si no
se indica se usará el espacio en blanco.
 * @param cadenas Array con las cadenas a unir
 * @return Devuelve una nueva cadena con el resultado de unir las cadenas
indicadas con el separador indicado
*/
func unirCadenas(separador:Character = " ", cadenas:[String]) → String
{
    var union:String = ""
    for cadena in cadenas
    {
        union += cadena
        union += String(separador)
    }
    return union
}

let resultado1 = unirCadenas(separador: "-", cadenas: ["valor1",
"valor2", "valor3"])
let resultado2 = unirCadenas(cadenas:["valor1", "valor2", "valor3"])

// Mostramos los resultados
print("El resultado 1 vale: \(resultado1)")
print("El resultado 2 vale: \(resultado2)")
```

Clases

Structs e inicializadores

El lenguaje de programación Swift soporta, entre muchos otros, el paradigma de la programación orientada a objetos, con lo que podremos definir nuestras propias clases con su determinada funcionalidad para resolver los problemas de una forma mucho más sencilla.

Para definir una clase, utilizaremos la palabra reservada `class` y escribiremos todo su código correspondiente dentro de un **bloque**.

Lo primero que tendremos que hacer es **definir las variables** de la clase, que **no tendrán ningún modificador de visibilidad**, ya que será el propio Swift el que se encargue de toda la seguridad de este tipo. Debemos **indicarle su tipo** cuando **declaremos** las variables.

Un elemento nuevo que nos ofrece Swift son las estructuras (**structs**), con las que podremos **agrupar una serie de variables**. Las estructuras y las clases son muy parecidas en Swift, pero con las **estructuras** podremos **definir las variables sin** necesidad de darles un **valor**.

Ejemplo de estructura y clase básica

```
struct calificaciones {  
    var nota1, nota2, nota3: Double  
}  
class Alumno {  
    let nombre:String    // Nombre del alumno  
    let edad: Int        //Edad del alumno  
    let notas: calificaciones    // Calificaciones del alumno  
}
```

Como ocurría en Java, **debemos crear los constructores** de la clase para poder crear objetos de ella. En Swift, los constructores se llaman **inicializadores**, y solo podremos tener **uno** con los parámetros que necesitemos.

Normalmente, se suele construir **un inicializador con todas las variables** de la clase.

Los inicializadores se crearán mediante la palabra reservada `init` y para acceder a las **variables de la propia clase**, deberemos usar `self`.

Ejemplo de inicializador en una clase

```
struct calificaciones {
    var nota1, nota2, nota3: Double
}
class Alumno {
    let nombre:String    // Nombre del alumno
    let edad: Int        //Edad del alumno
    let notas: calificaciones    // Calificaciones del alumno

    // Inicializador de la clase
    init(nombre: String, edad: Int, notas: calificaciones) {
        self.nombre = nombre
        self.edad = edad
        self.notas = notas
    }
}
```

Inicializadores de conveniencia

Si recordamos cuando estudiamos las clases en Java, podíamos declarar tantos constructores como necesitásemos, concretamente, tantos como combinaciones diferentes pudiéramos hacer con las variables de la clase.

Hemos visto en el punto anterior que únicamente podremos crear un inicializador en nuestras clases, pero ¿y si necesitamos **tener que declarar varios inicializadores** para poder cumplir con ciertas funcionalidades de un proyecto?

Esto también es posible realizarlo en Swift mediante lo que se conoce como inicializadores de conveniencia. Con este tipo de inicializadores, podremos crear tantos tipos de inicializadores como necesitemos y que nos permitan construir los objetos de nuestras clases de una forma mucho más cómoda.

El concepto de inicializador de conveniencia es muy simple, lo que estaremos creando son **inicializadores alternativos** que van a **terminar invocando al inicializador principal**, pero dando la posibilidad de **preprocesar los datos** antes de la **llamada al inicializador principal**.

Para crear un inicializador de conveniencia, deberemos utilizar las palabras `convenience init` seguidas **entre paréntesis de los parámetros** que vayamos a pasarle. Una vez que hayamos procesado los datos de la forma adecuada, tendremos que **llamar al inicializador principal** de la clase mediante `self.init`.

Ejemplo de inicializador de conveniencia

```
struct calificaciones {
    var nota1, nota2, nota3: Double
}

class Alumno {
    let nombre:String    // Nombre del alumno
    let edad: Int        //Edad del alumno
    let notas: calificaciones    // Calificaciones del alumno

    // Inicializador de la clase
    init(nombre: String, edad: Int, notas: calificaciones) {
        self.nombre = nombre
        self.edad = edad
        self.notas = notas
    }

    // Inicializador de conveniencia
    convenience init(nombre: String, edad: Int, notas1: Double, nota2:
Double, nota3: Double) {
        let notas = calificaciones(nota1: nota1, nota2: nota2, nota3:
nota3)
        self.init(nombre: nombre, edad: edad, notas: notas)
    }
}
```

Recordatorio sobre inicializadores

Recuerda que siempre que crees una clase en Swift, el **inicializador principal** de la misma deberá llevar como **parámetros todos los atributos**, para luego así poder tener libertad total de crear todos los **inicializadores de conveniencia que necesites**.

Heredando en Swift

El lenguaje de programación Swift es un lenguaje que **soporta la orientación a objetos**, y, como tal, va a poder soportar la herencia, pudiendo utilizarla como estamos acostumbrados de otros lenguajes (Java).

La herencia soportada por Swift es la **herencia simple**, que si recordamos, esto quería decir que una clase solo podrá heredar de otra, siendo una práctica mucho más segura que la herencia múltiple.

No obstante, Swift sí que nos proporciona una funcionalidad extra que puede sustituir a la herencia en algunas ocasiones. Esta es la extensión de clases.

La **extensión de clases** nos aporta la posibilidad de poder agregarle una funcionalidad extra a una clase ya creada, es decir, podemos hacer que esa clase “se extienda” y nos ofrezca más funcionalidades. Esto también se podría resolver **creando una clase que heredara de otra** e implementándole la **nueva funcionalidad** a la nueva clase. Con esto, tenemos que tener en cuenta que el uso de la herencia en Swift es **mucho más específico** que en lenguajes como Java.

Tuplas

Las tuplas son un concepto que nos proporciona el lenguaje de programación Swift, aunque también es cierto que las podemos encontrar en otros lenguajes.

Estas son una **agrupación de valores** que **no tienen por qué ser del mismo tipo** de datos, sino que podremos mezclar enteros con cadenas, con reales, etc.

Para poder declarar una variable del tipo tupla, tendremos que indicar todos sus elementos entre **paréntesis** y separados mediante comas:

```
var tupla = ("palabra", 100, -9.6)
```

En la tupla anterior, podemos ver que tenemos tres elementos dentro de ella, uno del tipo cadena, uno del tipo entero y el último del tipo real.

Las tuplas podremos mostrarlas mediante la instrucción `print`, presentando todos los valores de la misma al llevarlo a cabo.

```
var tupla = ("palabra", 100, -9.6)
print(tupla)
```

Resultado:

```
("palabra", 100, -9.5999999999999996)
```

También podremos acceder a los elementos de una tupla mediante su índice, por ejemplo, si accedemos a `tupla.2` del ejemplo anterior, estaremos accediendo al valor `-9.6`.

Accediendo al valor de una tupla por su índice

```
var tupla = ("palabra", 100, -9.6)
print(tupla.2)
```

Resultado:

```
-9.6
```

No solo vamos a poder crear tuplas de tres elementos, sino que podremos crearlas con los **elementos que necesitemos**, pudiendo acceder a sus valores para **consultarlos o modificarlos** mediante sus índices.

Otra forma de declaración de tuplas que podremos utilizar será **dándoles un nombre** a cada uno de sus **componentes**, de esta forma, cuando queramos **acceder** a un valor de un elemento de la misma, lo podremos hacer **mediante su nombre**, no mediante su índice.

Accediendo a un valor de una tupla por su nombre de parámetro

```
var tupla: (dato1: String, dato2: Int) = ("hola", 2020)
print(tupla.dato2)
```

Resultado:

2020

Por último, cabe destacar que podremos **devolver una tupla dentro de una función**, haciendo que dicha función pueda devolver todos los valores que necesitemos en forma de tupla.

Función que devuelve una tupla de dos valores

```
func funcion() → (valor1: String, valor2: Int) {
    return("Devuelvo", 4)
}
```

Enumerados en Swift

Exploraremos la definición y aplicación de enumerados en Swift. Aprenderemos cómo definirlos y utilizarlos de manera efectiva en nuestro código.

Contenido:

Definición de Enumerados:

Para definir un enumerado en Swift, utilizaremos la palabra reservada '`enum`', seguida del **nombre del enumerado**. Dentro del bloque de código del enumerado, emplearemos la palabra '`case`' para definir los **valores permitidos**.

Ejemplo Práctico:

Consideremos un enumerado llamado 'Movimiento', cuyos valores pueden ser: derecha, izquierda, arriba y abajo. Podremos declarar **variables de tipo 'Movimiento'** de la siguiente manera:

```
var movi = Movimiento.DERECHA // Declaración de una variable
```

Si deseamos asignarle el valor "ARRIBA", la sintaxis sería:

```
movi = Movimiento.ARRIBA
```

Visualización del Movimiento:

Utilizando la orden 'print', podemos mostrar el movimiento por pantalla. Una vez compilado correctamente nuestro ejemplo, procedemos a ejecutarlo.

Conclusión:

Hemos demostrado cómo definir, utilizar y visualizar enumerados en Swift, lo que proporciona claridad y organización a nuestro código, especialmente en casos donde hay conjuntos discretos de valores relacionados.

```
import Foundation
// Este enumerado representa las posibles direcciones de un movimiento
enum Movimiento
{
    case DERECHA, IZQUIERDA, ARRIBA, ABAJO
}
var movi = Movimiento.DERECHA
movi = Movimiento.ARRIBA
print("El movimiento es: \(movi)")
```


Test sobre diccionarios en Swift

¿Cuál es el resultado de ejecutar este código?

```
var tripulacion = ["Capitan": "Luis", "Carlos": "Simon"]  
tripulacion = [:]  
print(tripulacion.count)
```

Grupo de opciones de respuestas

- 9
- 5
- 0
- 12

El resultado de ejecutar este código sería:

0

Esto se debe a que:

- primero **se inicializa un diccionario llamado tripulacion** con dos elementos.
- Luego, se **redefine** el diccionario como un **diccionario vacío** con **[:]**.
- Finalmente, se imprime la cantidad de elementos en el diccionario, que en este caso es 0.

Por lo tanto, la respuesta correcta es **opción: 0**.

Test sobre arrays en Swift

¿Cuál será el resultado al ejecutar el siguiente código?

```
let nombres = ["Luis", "Isabel", "David", "María"]
if let nombre = nombres[1]
{
    print("Hola (nombre)")
}
```

Grupo de opciones de respuesta

- El código no compila.
- Se mostrará por pantalla el mensaje “Hola Isabel”.
- Se mostrará por pantalla el mensaje “Hola Luís”.
- Al ejecutar la aplicación se lanzará un error.

Solución:

El código no compila:

Parece que el error proviene de intentar utilizar la unión opcional (if let) en un **valor que no es opcional**. El código corregido sería el siguiente:

```
let nombres = ["Luis", "Isabel", "David", "María"]

if nombres.indices.contains(1) {
    let nombre = nombres[1]
    print("Hola \(nombre)")
}
```

En Swift, la [propiedad indices](#) de un arreglo proporciona un **rango de los índices válidos** para ese arreglo. Esto significa que puedes usar [indices.contains\(_\)](#) para verificar si un índice específico está **dentro del rango** válido del arreglo.

En este código, nombres.indices devuelve un rango de índices válidos para el arreglo nombres. Luego, utilizamos contains(1) para verificar si el índice 1 está dentro de ese rango. Si es así, accedemos al elemento del arreglo en ese índice y lo imprimimos. Esto evita el error.

Test sobre interpolación de cadenas en Swift

¿Cuál es el resultado al ejecutar este código?

```
func decirHola(a nombre: String) -> String
{
    return "Hola (nombre)!"
}
print("(decirHola(a: "Francis"))")
```

Grupo de opciones de respuesta

- Se mostrará por pantalla el mensaje “Hola (Francis)”.
- El código no compila
- Se mostrará por pantalla el mensaje “Hola Francis”.
- Al ejecutar la aplicación se lanzará un error.

Solución:

El código tal como está no compilará correctamente debido a un error de sintaxis. La forma correcta de imprimir el resultado de la función decirHola(a:) con el argumento "Francis" sería utilizando la **interpolación de cadenas**. Así que la opción correcta sería:

El código no compila.

Para corregirlo, deberías cambiar la última línea del código para que se imprima correctamente utilizando la interpolación de cadenas:

```
func decirHola(a nombre: String) -> String{
    return "Hola \(nombre)!"
}
print(decirHola(a: "Francis"))
```

dando como resultado:

Hola Francis!

Actividad sobre cálculo del número feliz

Se dice que un **número natural es feliz** si cumple que, si sumamos los cuadrados de sus dígitos y seguimos el proceso con los resultados obtenidos, finalmente obtenemos uno (1) como resultado.

Por ejemplo, el número 203 es un número feliz ya que

$$2^2 + 0^2 + 3^2 = 13 \quad 1^2 + 3^2 = 10 \quad 1^2 + 0^2 = 1$$

Se dice que un número es feliz de grado k si se ha podido demostrar que es feliz en un máximo de k iteraciones. Se entiende que una iteración se produce cada vez que se elevan al cuadrado los dígitos del valor actual y se suman. En el ejemplo anterior, 203 es un número feliz de grado 3 (además, es feliz de cualquier grado mayor o igual que 3).

- ✓ Realizar un programa que nos diga si un número es feliz.
- ✓ Realizar un programa que nos diga si un número es feliz y su grado.

Solución

```
func esNumeroFeliz(_ numero: Int) -> Bool {
    var num = numero
    var historial = Set<Int>()

    while num != 1 && !historial.contains(num) {
        historial.insert(num)
        var sumaCuadrados = 0
        while num > 0 {
            let digito = num % 10
            sumaCuadrados += digito * digito
            num /= 10
        }
        num = sumaCuadrados
    }

    return num == 1
}

func gradoNumeroFeliz(_ numero: Int) -> Int? {
    var num = numero
    var historial = Set<Int>()
    var grado = 0
```

```
while num != 1 && !historial.contains(num) {
    historial.insert(num)
    var sumaCuadrados = 0
    while num > 0 {
        let digito = num % 10
        sumaCuadrados += digito * digito
        num /= 10
    }
    num = sumaCuadrados
    grado += 1
}

return num == 1 ? grado : nil
}

// Ejemplo de uso
let numero = 203
if esNumeroFeliz(numero) {
    print("El número \(numero) es feliz.")
    if let grado = gradoNumeroFeliz(numero) {
        print("Grado de felicidad: \(grado)")
    }
} else {
    print("El número \(numero) no es feliz.")
}
```

Nota sobre ejecutables Switch

Cuando empezamos a programar en algún lenguaje, normalmente, tendemos a querer identificar cuanto antes la forma de crear un ejecutable de nuestros programas.

Si recordamos NetBeans, este nos ofrecía un botón con el cual podríamos crear un fichero .jar que era el equivalente a los ejecutables para la máquina virtual de Java. Igual pasaba con Android Studio, que nos ofrecía la posibilidad de crear una APK de nuestro proyecto para poder instalarlo en un dispositivo móvil, pero ¿ocurre lo mismo con Swift?

Swift es un **lenguaje interpretado**, por lo que, con este tipo de lenguajes, **no podremos crear un ejecutable** de los programas, ya que el funcionamiento de los mismos no lo permite.

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Introducción a MacOS y XCode

Introducción a MacOS

El sistema operativo Mac OS X fue **diseñado por Apple**.

Apple Inc. fue fundada en **1976** por **Steve Wozniak, Steve Jobs y Ron Wayne**, aunque la cara más conocida de Apple siempre fue **Steve Jobs**, ya que era quien anunciaba todos los productos y se encargaba del **marketing** y desarrollo de negocio de la empresa. No obstante, tanto Steve Wozniak como Ron Wayne fueron igual de relevantes y cruciales que Steve Jobs.

Apple creó el Apple I, un pequeño ordenador que ofrecían a sus, aún, pocos clientes, llegando a unas 200 ventas, aproximadamente. Fue con el dinero que obtuvieron con el Apple I con el que consiguieron desarrollar el Apple II para seguir con el Apple III, el **Apple III+** (que **corregía gran cantidad de desaciertos** de su antecesor), y el **Lisa**.

Pero no fue hasta la época de **1980** que no desarrollaron el **primer Macintosh**, el cual ya contaba con un sistema operativo con **interfaz gráfica**. Fue aquí cuando nació el **sistema operativo de Apple**.

Este fue el comienzo de los Macs, desarrollados íntegramente por la compañía Apple, la cual decidió llevar una fortísima política de **encapsulación y hermetismo**, y de desarrollar sus **propios ordenadores y sus sistemas** operativos, teniendo fuertemente acaparadas todas sus **patentes**.

Fue en el año **1999**, con la aparición de **MacOS X**, cuando decidieron dejar de desarrollar sus sistemas operativos desde cero y empezar a **basarse en UNIX** para ello.

Los primeros ordenadores Macintosh utilizaban un procesador llamado **PowerPC**, que, en aquella época, era de los que mejores prestaciones presentaban, hasta que **Intel** lo sobrepasó, y Apple, viendo esto, decidió pasar a desarrollar sus ordenadores con procesadores de Intel.

Los procesadores de los nuevos MacBook

Ya hemos comentado que Apple fabrica sus ordenadores, tanto de sobremesa como portátiles, basados en microprocesadores Intel.

Todo esto va a cambiar a partir de la salida de su **nuevo sistema operativo**, el llamado **Big Sur**, el cual estará optimizado para sus **propios procesadores**, dejando de colaborar con **Intel** después de tantos años.

Versiones de iOS y MacOS X

El **sistema operativo** que Apple usa para sus **smartphones** se llama **iOS**.

Apple suele anunciar sus **nuevas versiones** del sistema operativo **iOS** coincidiendo con el anuncio de algún **nuevo smartphone**. Estos lanzamientos se hacen en una **conferencia llamada WWDC**, celebrada en **California**, EEUU.

Las **versiones** que podemos encontrar hasta la fecha del sistema operativo iOS son las que podemos ver en la siguiente tabla.

Versiones del sistema operativo iOS

Versión	Fecha de salida
iOS 1	Junio 2007
iOS 2	Junio 2008
iOS 3	Junio 2009
iOS 4	Junio 2010
iOS 5	Octubre 2011
iOS 6	Septiembre 2012
iOS 7	Junio 2013
iOS 8	Septiembre 2014
iOS 9	Septiembre 2015
iOS 10	Septiembre 2016
iOS 11	Septiembre 2017
iOS 12	Junio 2018
iOS 13	Septiembre 2019
iOS 14	Septiembre 2020
iOS 15	Junio 2021
iOS 16	Septiembre 2022

Al igual que iOS, el sistema operativo MacOS X está desarrollado en versiones, las cuales son:

Versiones del sistema operativo MacOS X

1. Mac OS X **10.0 (Cheetah)**
2. Mac OS X 10.1 (**Puma**)
3. Mac OS X 10.2 (**Jaguar**)
4. Mac OS X 10.3 (**Panther**)
5. Mac OS X 10.4 (**Tiger**)
6. Mac OS X 10.5 (**Leopard**)
7. Mac OS X 10.6 (**Snow Leopard**)
8. Mac OS X 10.7 (Lion)
9. Mac OS X 10.8 (**Mountain Lion**)
10. **Mac OS X 10.9 (Mavericks)**
11. Mac OS X 10.10 (**Yosemite**)
12. Mac OS X **10.11 (El Capitán)**
13. Mac OS X 10.12 (**Sierra**)
14. **Mac OS X 10.13 (High Sierra)**
15. Mac OS X 10.14 (**Mojave**)
16. Mac OS X 10.15 (Catalina) (hasta Catalina son consecutivas, desde 10.0 a 10.15)
17. Mac OS X 11.6 (Big Sur)
18. Mac OS X 12.4 (**Monterey**)

¿Qué versión de MacOS X es mejor para desarrollar?

Es normal que cuando queremos adentrarnos en un sistema operativo que no conocemos, investiguemos un poco las distintas versiones que posee. Ocurre lo mismo con los sistemas operativos GNU/Linux y Windows, que tienen varias versiones e incluso distribuciones, en el caso de GNU/Linux.


MacOS X es algo distinto. Todas las versiones que existen no atienden a diferentes sistemas operativos, como ocurre con Windows XP, 7, 10..., sino que se corresponden a un único sistema operativo que ha ido **evolucionando** a lo largo de los años.

Debido a esto, lo más recomendable para empezar a usarlo y desarrollar aplicaciones será tener el **último sistema operativo que esté disponible**, ya que, además, las **nuevas actualizaciones** de las aplicaciones **no serán compatibles** con sistemas operativos de un **tiempo atrás**, ya que se quedarán **obsoletos** y no serán compatibles con ellas.










Atajos de teclado en MacOS

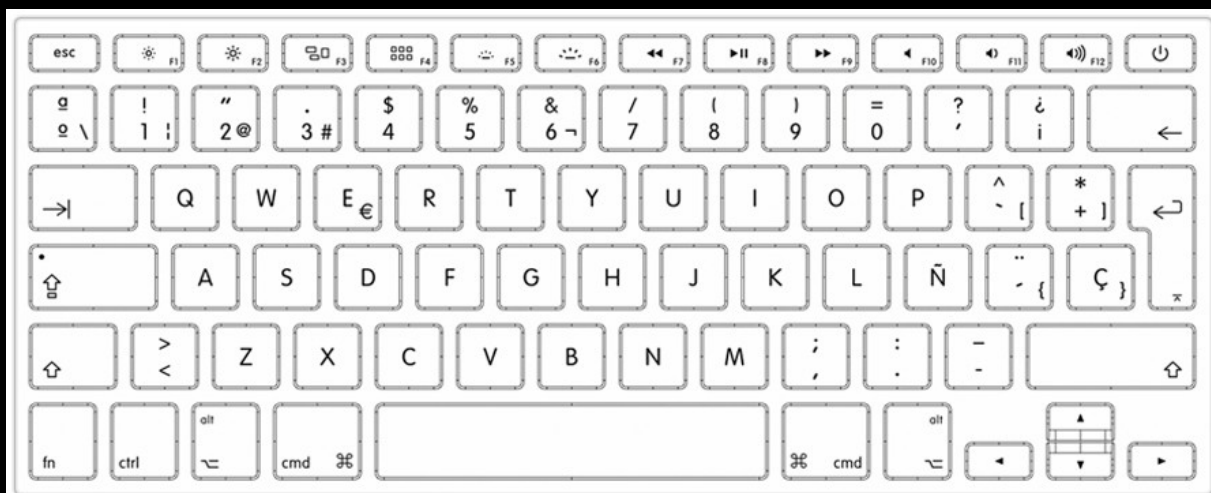
Todos conocemos los típicos atajos de teclado que nos ayudan a realizar las tareas de una forma mucho más simple, como el tan recurrido “control + c” para copiar o “control + z” para deshacer.

El sistema operativo MacOS X también posee una serie de atajos, pero **no son los mismos** a los que estamos acostumbrados si usamos GNU/Linux o Windows.

Para empezar, los teclados de Apple no son iguales que los teclados “normales” que todos estamos acostumbrados a utilizar, ya que estos disponen de una **tecla especial** llamada “**comando**”, cuyo símbolo es , que se usará para los atajos en este tipo de ordenadores.

Una lista de los atajos más comunes que vamos a usar es la siguiente:

- Comando () + C: Copiar (equivalente a Cntrl + C)
- Comando () + X: Cortar (equivalente a Cntrl + X)
- Comando () + V: Pegar (equivalente a Cntrl + V)
- Comando () + Z: Deshacer (equivalente a Cntrl + Z)
- Comando () + R: Esta combinación nos permitirá arrancar desde el **sistema recuperación** de MacOS.
- Opción (): Esta combinación nos permitirá arrancar el **Gestor de arranque**, que permite elegir **otros discos o volúmenes de arranque** si están disponibles.
- Mayúsculas (): Arranca en **modo seguro**.
- Comando () + S: Esta combinación nos permitirá arrancar en **modo de usuario único**. Es una combinación de teclas que requiere macOS **High Sierra** o una versión **anterior**.
- T: Arranca en modo de **disco de destino**.
- Comando () + V: Arranca en **modo detallado**.



Cerrar aplicaciones correctamente en MacOS X

Vamos a ver cómo podemos cerrar una aplicación correctamente en nuestro sistema operativo Mac.

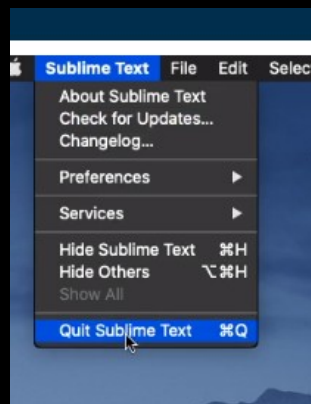
Aquí podemos ver que tenemos el escritorio y abajo nuestro **dock** con las aplicaciones que tenemos con acceso directo. Si pulsamos sobre alguna de ellas, por ejemplo, Sublime Text, esta aplicación se abrirá normalmente.

Si estamos acostumbrados a sistemas operativos como **Windows y GNU/Linux**, lo normal en estos sistemas operativos es cerrar la aplicación mediante el **botón de cerrar y se cerrará definitivamente**, pero en el sistema operativo Mac esto no es así.

En el sistema operativo Mac, si cerramos la aplicación directamente desde su botón de cerrar, ocurre lo siguiente: vemos que Sublime Text tiene un **punto debajo**, eso quiere decir que **se sigue ejecutando en segundo plano**. Con esto, lo que conseguimos es que si volvemos a seleccionar que **se abra**, lo haga **mucho más rápido**.



Si queremos cerrar la aplicación completamente, tendremos que venir a este **menú superior izquierdo**, que será el menú de nuestra aplicación que tenemos abierta. Seleccionar el nombre de la aplicación y la última opción siempre será cerrar la aplicación ('Quit' seguido del nombre del programa). Si pulsamos aquí, podemos ver que la aplicación se ha cerrado y debajo no tiene ningún punto blanco que indique que se está ejecutando en segundo plano.



Lenguajes de programación para aplicaciones en smartphones iOS

Ya sabemos, de unidades anteriores, que para desarrollar aplicaciones en Android, podemos utilizar los lenguajes Java y Kotlin, además del XML para la implementación de las interfaces gráficas.

Para el desarrollo de las aplicaciones para iOS, también podremos elegir entre dos lenguajes de programación, como son **Objective-C** y **Swift**, aunque **en el núcleo** de estos dispositivos se encuentra **Cocoa Touch**, la cual no es más que la **API** que **permite desarrollar las aplicaciones** para los dispositivos del sistema operativo de Apple.

Arquitectura de Apple

Objective-C		Swift	
Foundation	UIKit	...	
Cocoa Touch			
iOS			

Como podemos intuir de la imagen, la **API de Cocoa Touch** debe ser **enorme** y llena de **funcionalidades**, ya que es la encargada de hacer que **todo funcione** correctamente.

Los lenguajes de programación propiamente dichos, **Objective-C** y **Swift**, son los que nos van a permitir **“controlar” la API de Cocoa Touch** para que haga lo que necesitemos. (Se podría decir que Cocoa Touch es al sistema Mac lo que es “C” a los Sistemas Microsoft y Linux.)

En un principio, las aplicaciones se desarrollaban solo en Objective-C, aunque hace relativamente poco tiempo, se introdujo la posibilidad de poder desarrollar en **Swift**, un lenguaje mucho **más evolucionado y de alto nivel que el Objective-C**. Esto no quiere decir que Objective-C no se use en la actualidad, todo lo contrario, todavía hay multitud de aplicaciones que se desarrollan o están desarrolladas en este lenguaje, por la simple razón de que al llevar tanto tiempo utilizándose, hay gran cantidad de tareas que están desarrolladas en él y se pueden reutilizar.

No obstante, el **futuro** del desarrollo de aplicaciones para smartphones Apple es el lenguaje **Swift**, ya que ofrece muchas más **ventajas respecto a Objective-C**. Algunas de ellas:

- El código es mucho más **conciso**.
- Gestiona **automáticamente** la **memoria**.
- Ofrece un **tipado fuerte** de datos.

Instalación de XCode

XCode es el entorno de desarrollo oficial que nos ofrece Apple para poder desarrollar aplicaciones tanto para el sistema operativo **MacOS X** como para sus **smartphones**. Tenemos que tener en cuenta que no podremos desarrollar aplicaciones para los sistemas operativos de Apple **en ningún otro programa que no sea XCode**.

Para poder instalar XCode, necesitamos cumplir los siguientes requisitos:

1. Un **ordenador Mac** con la versión del sistema operativo actualizada a **la más reciente**.
2. Una **cuenta** creada y validada de **iTunes**.
3. La aplicación **App Store** instalada en nuestro Mac, la cual ya viene por defecto en la instalación, y con la **cuenta de iTunes** con la sesión **iniciada**.

Cuando cumplamos los requisitos anteriores, ejecutamos la App Store.

Una vez iniciada, la localizamos con el buscador de apps mediante su nombre: XCode.

Una vez que hayamos seleccionado XCode en la App Store, deberemos pulsar en "Obtener" para que comience la descarga del programa y su instalación.

El proceso de **instalación** de XCode es **muy largo**, ya que este entorno de desarrollo es bastante completo y deberá descargarse en su totalidad antes de poder instalarse.

¿Dónde instalar XCode?

Ya conocemos ciertas características sobre la compañía Apple, desarrolladora de los iPhones, MacBook, etc., y un aspecto que no deja indiferente a nadie que quiera iniciarse en su sistema operativo, o desarrollar apps para sus dispositivos móviles, es el hermetismo del que se dispone en la compañía.

No podremos instalar sus sistemas operativos en otros PC que no sean su marca, por incompatibilidades de hardware. Estos problemas no solo se limitan al aspecto del sistema operativo, sino que con sus aplicaciones ocurre lo mismo, y XCode no es una excepción.

Si queremos instalar XCode para aprender a desarrollar aplicaciones para los smartphones de Apple, no podremos hacerlo a menos que dispongamos de un ordenador Apple, ya que XCode no está disponible para ningún otro sistema operativo.

Creación de un proyecto en XCode

Una vez que ya tenemos instalado XCode en un Mac, estamos en condiciones de crear un nuevo proyecto. Para ello pulsamos sobre la opción

“Create a new Xcode Project”

y seleccionamos el tipo de app

“Single View App”

que es una aplicación vacía.

Aquí tendremos que cumplimentar los siguientes [datos](#):

- **Product Name:** Este es el nombre con el que vayamos a nombrar a la app.
- **Organization Name:** Este es el nombre del **desarrollador** de apps; en nuestro caso, podemos poner nuestro nombre.
- **Organization Identifier:** Este es el **identificador** del desarrollador de apps.
- **Language:** Lenguaje en el que va a estar programada la app. Podremos elegir entre Objective-C y Switt. Nosotros elegiremos siempre Swift.

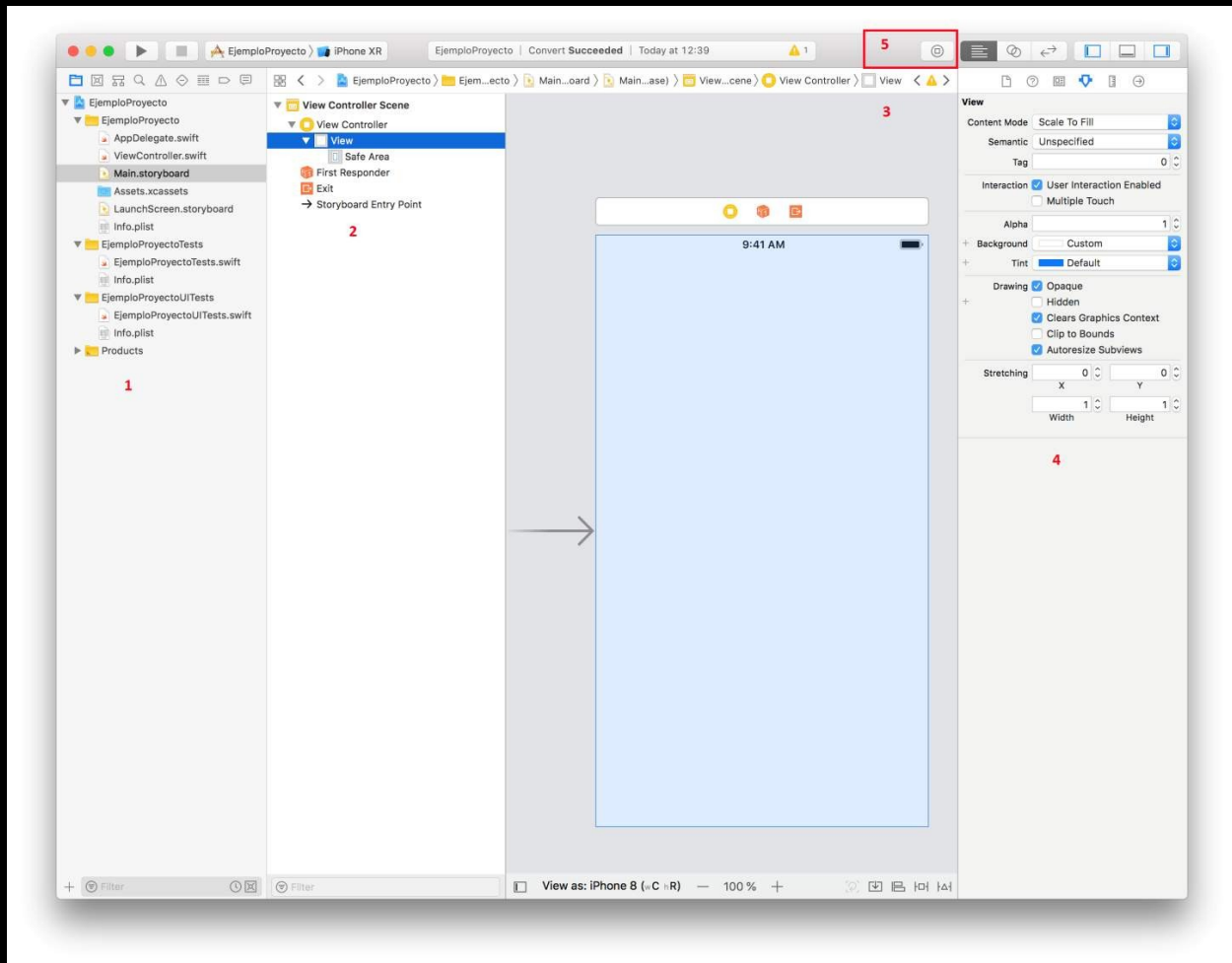
Una vez cumplimentado todo esto, pulsamos el botón

“Next”

y seleccionamos donde queremos guardar el nuevo proyecto.

Una vez creado el proyecto, pulsamos en **Main.storyboard** y podremos ver una pantalla como la de la siguiente figura:

Proyecto vacío en Xcode



1. En esta área (de la izquierda), tenemos la **estructura** con todos los ficheros .swift y carpetas que componen nuestro proyecto.
2. En esta zona (a continuación a la derecha de la estructura), se sitúan los **elementos gráficos** que componen cada una de las vistas de nuestra app.
3. Esta sección contiene las vistas de nuestra app en “**vista diseño**”.
4. En el margen **derecho**, se sitúan las **propiedades de los elementos** gráficos y vistas de nuestra app.
5. En esta posición (visible el cambiador de posición en la parte superior), tenemos la **paleta de elementos** gráficos que podemos insertar en nuestra app.

Creación y ejecución de un proyecto en XCode

Vamos a ver cómo podemos crear un proyecto de Xcode y sus partes.

1. **Abrir Xcode:** Para abrir Xcode, simplemente necesitaremos hacer clic en su icono y el programa se abrirá.

2. **Crear un nuevo proyecto:** Una vez abierto Xcode, vamos a crear un nuevo proyecto siguiendo estos pasos:

- Pulsamos en "Crear un **nuevo proyecto**" y seguimos el asistente.
- Elegimos la opción de "**Aplicación vacía**".
- Introducimos el nombre de nuestra aplicación (por ejemplo, "Aplicación Prueba").
- Seleccionamos el lenguaje (en este caso, Swift) y la interfaz (Storyboard o SwiftUI). Nosotros elegiremos Storyboard.
- Introducimos la información de la organización y pulsamos en "Siguiente".
- Elegimos la ubicación donde queremos guardar el proyecto (por ejemplo, en el escritorio), luego pulsamos en "Crear".

3. **Explorar el proyecto:** Una vez creado el proyecto, podemos explorar sus partes:

- Abrimos el "Main Storyboard" para ver la interfaz gráfica del proyecto, que inicialmente será una pantalla vacía.
- Podemos ver tres partes principales:
 - El árbol de archivos y carpetas del proyecto.
 - La interfaz gráfica donde previsualizamos el diseño de la aplicación.
 - El panel de propiedades de los componentes de la interfaz.

4. **Editar el código:** Para editar el código, simplemente hacemos clic en el archivo correspondiente y se abrirá en el editor de código, que estará en Swift.

5. **Ejecutar la aplicación:** Para ejecutar la aplicación en un emulador, seguimos estos pasos:

- Seleccionamos el emulador en el que queremos ejecutar la aplicación (por ejemplo, iPhone 8).
- Pulsamos el botón de ejecutar y se abrirá un emulador donde se instalará y ejecutará la aplicación.
- Esperamos a que se inicie y cargue la aplicación en el emulador.

¡Listo! Ahora tenemos nuestra aplicación ejecutándose en el emulador.

Reflexión

En el mundo de las aplicaciones móviles, ya sabemos que existen dos gigantes: Android y Apple.

Las aplicaciones Android ya las dominamos de las unidades anteriores, pero si nos estamos planteando dedicarnos a ello profesionalmente, ¿qué ocurre con las aplicaciones para Iphone?, ¿merecerá la pena desarrollar aplicaciones para este sistema, dado el tiempo que requerirá aprender, y la inversión que se necesitaría llevar a cabo?

Es verdad que ambos sistemas operativos ofrecen un nicho de mercado muy amplio por separado, pero también cabe destacar, que cuantos más servicios pueda ofrecer una empresa de desarrollo, más posibilidades de negocio tendrá. Por tanto, si nos estamos planteando **crear una empresa de desarrollo de apps para móviles**, sería muy recomendable saber hacerlo para las **dos plataformas**.

Por otra parte, si no es a través de una empresa propia, sino que salimos al mundo laboral a buscar trabajo en otras empresas, un **programador polivalente**, que domine **los dos sistemas**, tendrá muchas **más posibilidades** de encontrar un empleo, así que podemos deducir que estaría bien dominar las dos marcas, aunque estemos **especializados más en una** que en otra.