school everis

FORMAMOS PROFESIONALES

Módulo 8

Código: 489

Programación multimedia y dispositivos móviles

Técnico Superior en Desarrollo de Aplicaciones Multiplataforma

school everis

FORMAMOS PROFESIONALES

UNIDAD 3

Programación de aplicaciones para dispositivos móviles. Lógica y conexiones

Contenido teórico

Índice

1. Objetivos generales de la unidad
2. Competencias y contribución en la unidad
3. Contenidos conceptuales y procedimentales
4. Evaluación
5. Distribución de los contenidos
6. Sesiones

1. Objetivos generales de la unidad

Objetivos curriculares

school everis

FORMAMOS PROFESIONALES

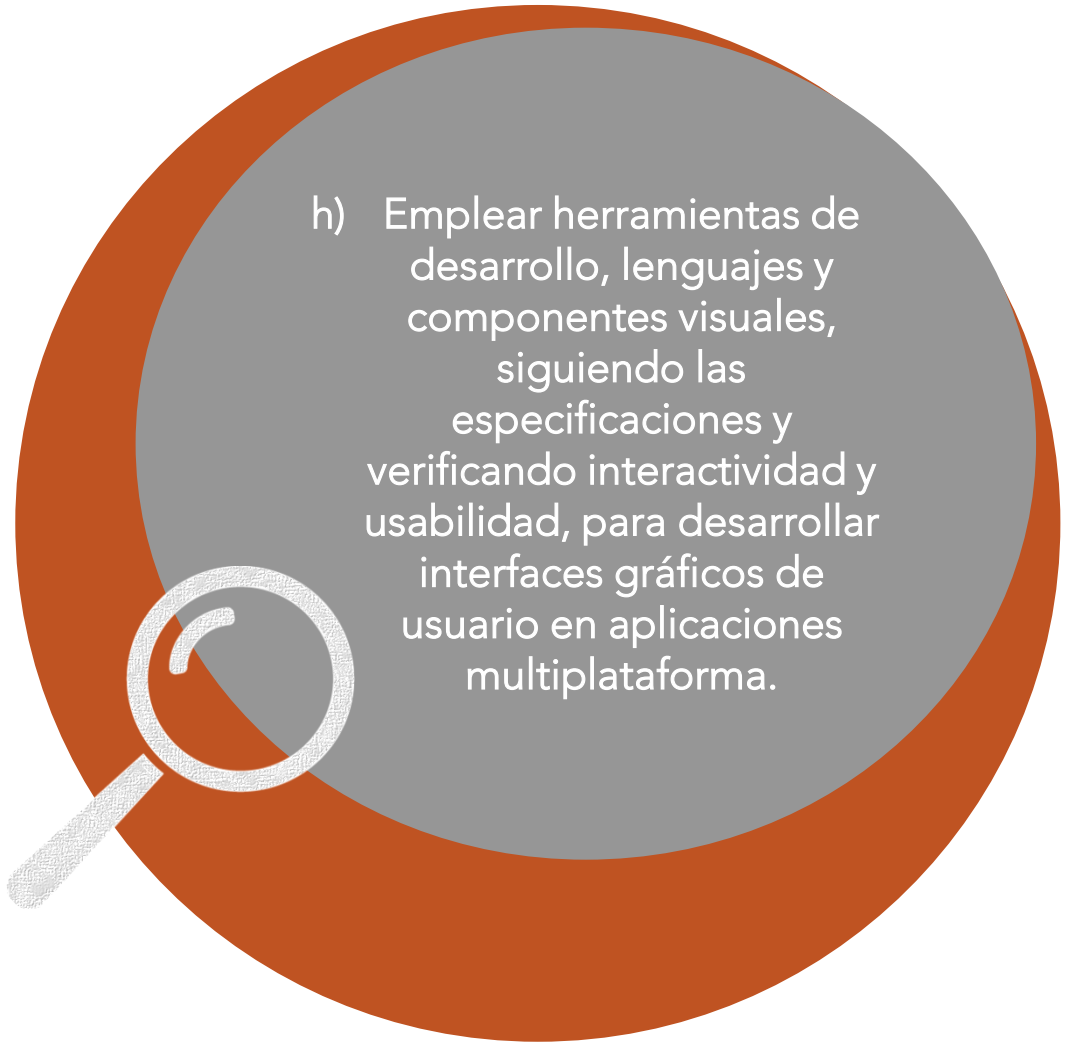
1. Describir la evolución, características y limitaciones de los dispositivos para el desarrollo de aplicaciones móviles.

2. Dar una visión global de la Computación Ubicua.

3. Analizar las tecnologías disponibles, los lenguajes y los entornos integrados de trabajo y compilación.

4. Proporcionar los conceptos básicos y las herramientas que permitan al alumno adquirir las competencias necesarias para trabajar con los distintos entornos de desarrollo de aplicaciones para dispositivos móviles.

2. Competencias y contribución en la unidad



Analizando los diferentes elementos disponibles para la creación de interfaces gráficas y usándolos para la composición de una pantalla con la que el usuario pueda interactuar.

3. Contenidos

school everis

FORMAMOS PROFESIONALES

CONTENIDOS CONCEPTUALES	CONTENIDOS PROCEDIMENTALES
• SQLite • DOM • ContentProvider • WebView • NFC • Bluetooth • RFCOMM	• Descubrimiento de servicios. • Bases de datos y almacenamiento. • Persistencia. • Modelo de hilos. • Comunicaciones: clases asociadas. Tipos de conexiones. • Gestión de la comunicación inalámbrica. • Búsqueda de dispositivos. • Búsqueda de servicios. • Establecimiento de la conexión. Cliente y servidor. • Envío y recepción de mensajes texto. Seguridad y permisos. • Envío y recepción de mensajería multimedia. Sincronización de contenido. Seguridad y permisos. • Manejo de conexiones HTTP y HTTPS. • Complementos de los navegadores para visualizar el aspecto de un sitio web en un dispositivo móvil. • Pruebas y documentación.

4. Evaluación

school everis FORMAMOS PROFESIONALES

a) Se han utilizado las clases necesarias para establecer conexiones y comunicaciones HTTP y HTTPS.

b) Se han utilizado las clases necesarias para establecer conexiones con almacenes de datos garantizando la persistencia.

c) Se han realizado pruebas de interacción usuario-aplicación para optimizar las aplicaciones desarrolladas a partir de emuladores.

d) Se han documentado los procesos necesarios para el desarrollo de aplicaciones.

Sesión 1. Contenidos teóricos

school everis FORMAMOS PROFESIONALES

1. INTENTS

Un Intent es un objeto de acción que puedes usar para solicitar una acción de otro componente de la aplicación. Aunque las *intents* facilitan la comunicación entre componentes de muchas maneras, existen tres casos de uso fundamentales:

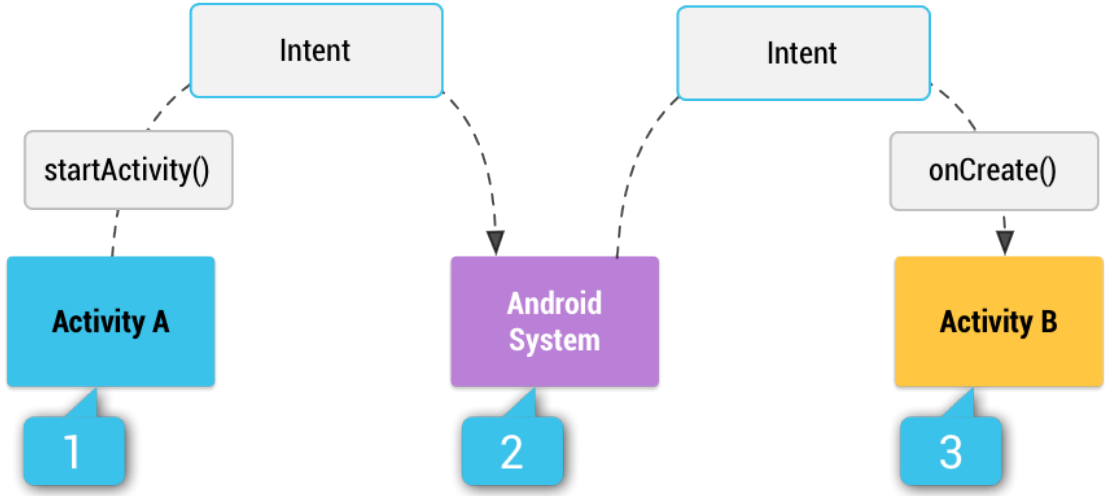
- Para comenzar una actividad
- Para iniciar un servicio
- Para entregar un mensaje

Además, se distinguen dos tipos:

- **Explícitas**: que especifican qué componente se debe iniciar mediante su nombre.
- **Implícitas**: no se nombre el componente específicamente.

1. INTENTS

Ilustración de la forma en que se entrega una intent implícita mediante el sistema para iniciar otra actividad.
[1] La actividad A crea una Intent con una descripción de acción y la pasa a startActivity().
[2] El sistema Android busca en todas las apps un filtro de intents que coincida con la intent. Cuando se encuentra una coincidencia,
[3] el sistema inicia la actividad coincidente (actividad B) invocando su método onCreate() y pasándolo a la Intent.



Sesión 1. Contenidos teóricos

1. INTENTS

Para comenzar una actividad:

- Una Activity representa una única pantalla en una aplicación. Puedes iniciar una nueva instancia de una Activity pasando una Intent a startActivity(). La Intent describe la actividad que se debe iniciar y contiene los datos necesarios para ello.
- Si deseas recibir un resultado de la actividad cuando finalice, llama a startActivityForResult(). La actividad recibe el resultado como un objeto Intent separado en el callback de onActivityResult() de la actividad.

1. INTENTS

Para iniciar un servicio:

- Un Service es un componente que realiza operaciones en segundo plano sin una interfaz de usuario. Puede iniciar un servicio para realizar una operación única (como descargar un archivo) pasando una Intent a startService(). La Intent describe el servicio que se debe iniciar y contiene los datos necesarios para ello.
- Si el servicio está diseñado con una interfaz cliente-servidor, puedes establecer un enlace con el servicio de otro componente pasando una Intent a bindService(). Para obtener más información, consulte la guía Servicios.

1. INTENTS

Para entregar un mensaje:

- Un mensaje es un aviso que cualquier aplicación puede recibir. El sistema entrega varios mensajes de eventos del sistema, como cuando el sistema arranca o el dispositivo comienza a cargarse.
- Puedes enviar un mensaje a otras apps pasando una Intent a sendBroadcast(), sendOrderedBroadcast() o sendStickyBroadcast().

2. TIPOS DE INTENTS

- **Intents explícitas**: especifican qué componente se debe iniciar mediante su nombre (el nombre de clase completamente calificado). Usualmente, el usuario usa una intent explícita para iniciar un componente en su propia aplicación porque conoce el nombre de clase de la actividad o el servicio que desea iniciar. Por ejemplo, puede utilizarla para iniciar una actividad nueva en respuesta a una acción del usuario o iniciar un servicio para descargar un archivo en segundo plano.
- **Intents implícitas**: no se nombra el componente específicamente; pero, en cambio, se declara una acción general para realizar, lo que permite que un componente de otra aplicación la maneje. Por ejemplo, si desea mostrar al usuario una ubicación en un mapa, puede usar una intent implícita para solicitar que otra aplicación capaz muestre una ubicación específica en un mapa.

3. INFORMACIÓN DE UNA INTENT

- **Nombre del componente:** nombre del componente que debe iniciar.
- **Acción:** una string que especifica la acción genérica que se debe realizar (como ver o elegir).
- **Datos:** El URI que hace referencia a los datos en los que se debe realizar la acción el tipo de MIME de esos datos.
- **Categoría:** Una string que contiene información adicional sobre el tipo de componente que la intent debe manejar.
- **Extras:** Pares de valores clave que tienen información adicional necesaria para lograr la información solicitada.
- **Indicadores**: Metadatos.

4. EJEMPLOS

Ejemplo de una intent explícita:

- Una intent explícita es una intent que se usa para iniciar un componente específico de la aplicación, como una actividad o un servicio particular en la aplicación. Para crear una intent explícita, define el nombre de componente para el objeto Intent, todas las otras propiedades de la intent son opcionales.
- Por ejemplo, si creaste un servicio en tu app denominado DownloadService, diseñado para descargar un archivo de la Web, puedes iniciarlo con el siguiente código:

```
// Executed in an Activity, so 'this' is the Context
// The fileUrl is a string URL, such as "http://www.example.com/image.png"
Intent downloadIntent = new Intent(this, DownloadService.class);
downloadIntent.setData(Uri.parse(fileUrl));
startService(downloadIntent);
```

4. EJEMPLOS

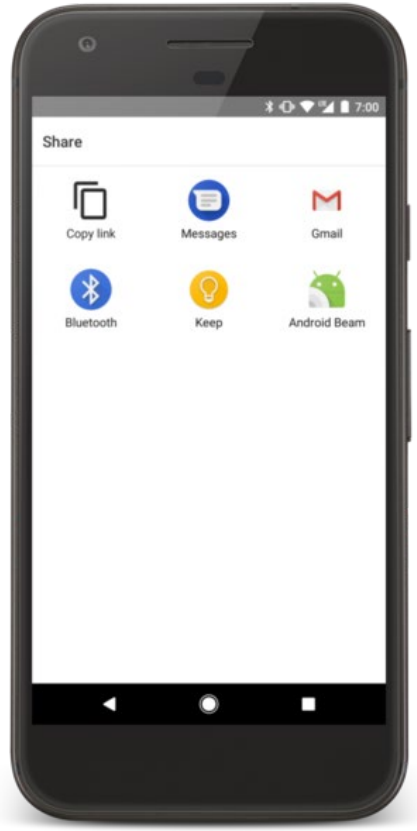
Ejemplo de una intent implícita:

- Una intent implícita especifica una acción que puede invocar cualquier aplicación en el dispositivo que pueda realizar la acción. El uso de una intent implícita es útil cuando la aplicación no puede realizar la acción pero otras aplicaciones probablemente sí, y tu desearas que el usuario eligiera qué aplicación usar.
- Por ejemplo, si tiene contenido que deseas que el usuario comparta con otras personas, crea una intent con la acción ACTION_SEND y agrega extras que especifiquen el contenido para compartir. Cuando llama a startActivity() con esa intent, el usuario puede elegir una aplicación mediante la cual compartir el contenido.

4. EJEMPLOS

```
// Create the text message with a string
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");

// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(sendIntent);
}
```



5. CONSERVACIÓN DEL ESTADO DE UNA ACTIVIDAD

Cuando una actividad pasa al estado en pausa o parada el estado de la actividad se mantiene (sigue en memoria)

- Cualquier cambio realizado en ella por el usuario se mantiene (y estará disponible cuando la actividad vuelva al estado en ejecución)

Cuando una actividad es destruida por el sistema porque la cantidad de memoria libre es demasiado baja, no se mantiene su estado en memoria

- Cuando el usuario solicita arrancarla de nuevo el sistema tiene que crear una nueva actividad (con su estado inicial) y luego recuperar (recrear) el estado que tenía cuando fue destruida
- Es necesario recuperar este estado porque el usuario no debe ser consciente de que la actividad ha sido destruida (sin que él lo solicitase)

5. CONSERVACIÓN DEL ESTADO DE UNA ACTIVIDAD

Para guardar el estado de una actividad antes de que sea destruida (y eliminada de memoria), se usa el método onSaveInstanceState(Bundle savedInstanceState)

- Este método es llamado antes de que la actividad pueda ser destruida (justo después de finalizar el método onPause())
- El código de este método debe incluir todas las tareas necesarias para salvar el estado de la actividad.
- Usa un parámetro del tipo Bundle en el que se salvará el estado mediante pares de clave-valor:
  - Método putString(String key, String value)
  - Método putInt(String key, int value)
- El estado de los widgets de entrada de datos (*checkboxes*, *radioButtons*, *editTexts*…) es salvado automáticamente por el método onSaveInstanceState de la superclase Activity.

5. CONSERVACIÓN DEL ESTADO DE UNA ACTIVIDAD

Para recuperar el estado de una actividad destruida por el sistema existen dos opciones:

- Usar el método onCreate(Bundle savedInstanceState)
  - Si la actividad está siendo recreada después de que el sistema la haya destruido, el parámetro de tipo Bundle será distinto de null
  - Si la actividad se crea por el usuario sin que haya sido previamente destruida por el sistema (la habrá finalizado anteriormente el usuario), el parámetro de tipo Bundle será null
  - Es necesario comprobar el valor de este parámetro antes de intentar usarlo
- Usar el método onRestoreInstanceState(Bundle savedInstanceState)
  - Este método solo es llamado si hay estado que recuperar de la actividad (cuando es llamado, su parámetro siempre será distinto de null)

Para recuperar el estado almacenado en un objeto de tipo Bundle se utilizan sus métodos getString(String key), getInt(String key)…

5. CONSERVACIÓN DEL ESTADO DE UNA ACTIVIDAD

Orden de ejecución de los métodos del ciclo de vida de una actividad:

- Al crear la aplicación: onCreate → onStart → onResume
- Cuando desde la actividad A se crea una actividad b:
  - A ejecuta onPause
  - B ejecuta onCreate, onStart y onResume
  - A ejecuta onSaveInstanceState, si A está completamente ocultada por B, A ejecuta onStop.
- Cuando se vuelve a la actividad anterior usando el botón "atrás" (desde la actividad B se pasa a la A)
  - B ejecuta onPause
  - A ejecuta onRestart, onStart y onResume
  - B ejecuta onStop y onDestroy

1. ALMACENAMIENTO PERSISTENTE

El problema de almacenar la información de una aplicación solo en variables es que en el momento en el que la aplicación es destruida, dejan de existir. Por ello, es habitual que se necesite almacenar información de forma permanente.

Las alternativas habituales para mantener información son:

- **Preferencias**: mecanismo liviano que permite almacenar y recuperar datos primitivos en forma de clave/valor.
- **Ficheros**: se pueden almacenar ficheros en la memoria interna del dispositivo o en un medio de almacenamiento removible.
- **XML**: es uno de los estándares más utilizados en la actualidad para codificar información. Disponemos de liberarías SAX y DOM para manipular estos ficheros desde Android.
- **Bases de datos**: Android tiene soporte para SQLite, de modo que se pueden crear y usar bases de datos SQLite de forma muy sencilla y con toda la potencia que nos ofrece el lenguaje SQL.
- **Proveedores de contenido**: un proveedor de contenido es un componente opcional de una aplicación que expone el acceso de lectura/escritura de sus datos a otra aplicaciones.
- **Internet**: nube para almacenar y recuperar datos.

2. PREFERENCIAS

La clase de SharedPreferences ofrece un marco general que permite guardar y recuperar pares clave-valor persistentes de tipos de datos primitivos. Se puede usar SharedPreferences para guardar todos los tipos de datos primitivos: booleanos, elementos flotantes, valores enteros, valores largos y strings. Estos datos se conservarán de una sesión de usuario a otra (incluso si la aplicación finaliza).
Si se desea obtener un objeto de SharedPreferences para la aplicación, se puede usar uno de los dos métodos siguientes:

- getSharedPreferences(): esta opción se usa si se necesitan varios archivos de preferencias identificados por nombre, que se especifican con el primer parámetro.
- getPreferences(): esta opción se usa si solo necesitas un archivo de preferencias para tu actividad. Debido a que este será el único archivo de preferencias para la actividad, no se debe proporcionar un nombre.

2. PREFERENCIAS

Para escribir valores, se hace lo siguiente:

- Llamar a edit() para obtener un SharedPreferences.Editor.
- Agregar valores con métodos, como putBoolean() y putString().
- Confirmar los nuevos valores con commit().

Para leer valores, usa métodos de SharedPreferences, como getBoolean() y getString().

El siguiente ejemplo describe cómo guarda una preferencia para el modo de presión silencioso en una calculadora.

2. PREFERENCIAS

```
public class Calc extends Activity {
    public static final String PREFS_NAME = "MyPrefsFile";

    @Override
    protected void onCreate(Bundle state){
       super.onCreate(state);
       . . .

       // Restore preferences
       SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
       boolean silent = settings.getBoolean("silentMode", false);
       setSilent(silent);
    }

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

      // We need an Editor object to make preference changes.
      // All objects are from android.context.Context
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor = settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      // Commit the edits!
      editor.commit();
    }
}
```

Sesión 3. Contenidos teóricos

school everis FORMAMOS PROFESIONALES

3. FICHEROS

- Se pueden guardar archivos directamente en el almacenamiento interno del dispositivo. De forma predeterminada, los archivos que se guardan en el almacenamiento interno son privados para la aplicación y otras aplicaciones no pueden tener acceso a ellos (tampoco el usuario). Cuando el usuario desinstala la aplicación, estos archivos se quitan.

- Además, todo dispositivo compatible con Android admite un "almacenamiento externo" compartido, que puedes usar para guardar archivos. Puede ser un medio de almacenamiento extraíble (como una tarjeta SD) o un medio almacenamiento interno (no extraíble). Los archivos guardados en el almacenamiento externo pueden ser leídos por cualquier usuario y este puede modificarlos cuando habilita el almacenamiento masivo de USB para transferir archivos a un ordenador.

3. FICHEROS

Para crear y escribir un archivo privado en el almacenamiento interno, hay que hacer lo siguiente:

- Llamar a openFileOutput() mediante el nombre del archivo y el modo de operación. Esto muestra un FileOutputStream.
- Realizar operaciones de escritura en el archivo con write().
- Cerrar el flujo con close().

```
String FILENAME = "hello_file";
String string = "hello world!";

FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

3. FICHEROS

Para leer un archivo desde el almacenamiento interno, se hace lo siguiente:

- Llamar a openFileInput() y usar el nombre del archivo que se desee leer. Esto muestra un FileInputStream.
- Leer los bytes del archivo con read().
- Cerrar el flujo con close().

```
String fichero = "fichero_memInt.txt";
try {
   BufferedReader fin = new BufferedReader(new InputStreamReader(
      openFileInput(fichero)));
   String texto = fin.readLine();
   fin.close();
}
catch (Exception ex) {
   Log.e("DADM: Ficheros", "Error al leer fichero desde memoria
interna");
}
```

3. FICHEROS

En el caso de almacenamiento externo, un ejemplo sería:

```
String texto = "texto de prueba";
String fichero = "fichero_memExt.txt";
try{
   File tarjeta = Environment.getExternalStorageDirectory();
   File f = new File(tarjeta.getAbsolutePath(), fichero);
   OutputStreamWriter fout = new OutputStreamWriter(new
FileOutputStream(f));
   fout.write(texto);
   fout.close();
}
catch (Exception ex){
   Log.e("DADM: Ficheros", "Error al escribir fichero a tarjeta SD");
}
```

Hay que especificar el permiso para escritura de almacenamiento externo en AndroidManifest.xml.

4. BASES DE DATOS

Android ofrece compatibilidad total con las bases de datos SQLite. Cualquier clase dentro de la aplicación, no fuera de ella, será posible acceder por nombre a cualquier base de datos que crees.

El método recomendado para crear una nueva base de datos SQLite consiste en crear una subclase de SQLiteOpenHelper y anular el método de onCreate(), en el cual puedes ejecutar un comando de SQLite a fin de crear tablas en la base de datos.

4. BASES DE DATOS

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 2;
    private static final String DICTIONARY_TABLE_NAME = "dictionary";
    private static final String DICTIONARY_TABLE_CREATE =
                "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
                KEY_WORD + " TEXT, " +
                KEY_DEFINITION + " TEXT);";

    DictionaryOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DICTIONARY_TABLE_CREATE);
    }
}
```

5. PROVEEDORES DE CONTENIDO

Un proveedor de contenidos (Content Provider, CP) es el mecanismo que proporciona Android para compartir información entre aplicaciones. Así, este mecanismo permite acceder a información que proporcionan otras aplicaciones, por ejemplo, la lista de contactos, la aplicación de mensajería, el calendario/agenda, etc. E igualmente, compartir nuestra información con otras aplicaciones.

Para crear un proveedor de contenidos:

1. Se definen el contenedor de datos (base de datos…) y todas las constantes necesarias (URIs, nombres de las columnas…)
2. Se crea una clase que extienda de la clase ContentProvider.
3. Se implementan los métodos query(), insert(), update(), delete() y getType().
4. Se declara el nuevo proveedor de contenidos en el fichero AndroidManifest.xml.

1. HILOS

Un hilo es una unidad de ejecución asociada a una aplicación. Es la estructura de la programación concurrente, la cual tiene como objetivo dar la percepción al usuario que el sistema que ejecuta realiza múltiples tareas a la vez.

Aunque los hilos se benefician de las tecnologías multinucleos y multiprocesamiento, no significa que una arquitectura simplista no se beneficie de la creación de hilos.

Cuando se construye una aplicación Android, todos los componentes y tareas son introducidos en el hilo principal o hilo de UI (UI Thread). Hasta el momento hemos trabajado de esta forma, ya que las operaciones que hemos realizado toman poco tiempo y no muestran problemas significativos de rendimiento visual en nuestros ejemplos.

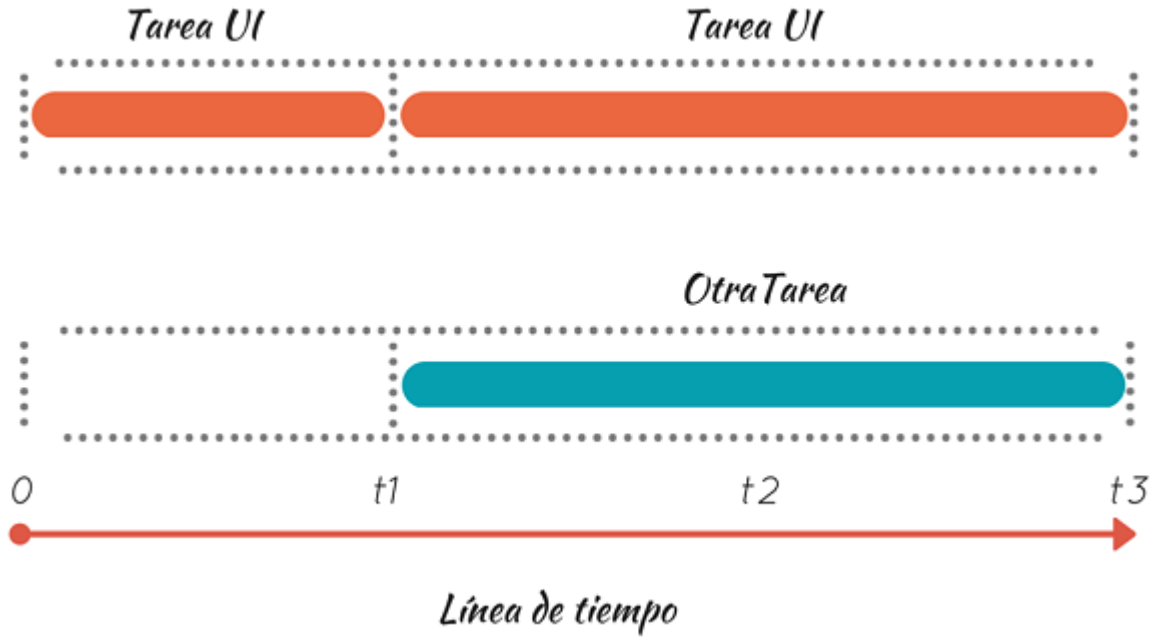
1. HILOS

¿Qué pasaría si se intenta cargar una imagen jpg de 3MB desde un servidor externo vía HTTP en tu aplicación?

- Si la conexión es rápida, tal vez nada.
- Pero para conexiones lentas esto tomara algunos segundos. Esto arruina la fluidez visual, lo que no gusta a los usuarios, pudiendo llegar a eliminar la aplicación. A nivel computacional, se puede apreciar de la siguiente forma:

1. HILOS

El camino correcto es renderizar la interfaz de la aplicación y al mismo tiempo ejecutar en segundo plano la otra actividad para continuar con la armonía de la aplicación y evitar paradas inesperadas. Es aquí donde entran los hilos, porque son los únicos que tienen la habilidad especial de permitir al programador generar concurrencia en sus aplicaciones y la sensación de multitareas ante el usuario.



1. HILOS

Se ha creado un nuevo hilo donde se ejecuta la tarea en el mismo intervalo de tiempo [t1, t2], pero esta vez el tiempo de ejecución de la tercera tarea UI se extendió debido a que se realizarán pequeños incrementos entre al segundo y tercera tarea. Aunque el tiempo empleado es el mismo, la respuesta ante el usuario simula una aplicación limpia.

Recuerda que existen dos tipos de procesamientos de tareas, Concurrencia y Paralelismo. La concurrencia se refiere a la existencia de múltiples tareas que se realizan simultáneamente compartiendo recursos de procesamiento.

El paralelismo es la ejecución de varias tareas al tiempo en distintas unidades de procesamiento, por lo que es mucho más rápido que la concurrencia.

Sesión 6. Contenidos teóricos

school everis FORMAMOS PROFESIONALES

1. HILOS

Para la llamada asíncrona de funciones, existe la clase AsyncTask. Un ejemplo de su uso sería crear la siguiente clase:

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

Y llamarla de la siguiente manera:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

2. MODELO CLIENTE-SERVIDOR EN ANDROID

El modelo cliente – servidor en Android funciona de manera similar que en Java. Puesto que ya está visto en otros módulos, vamos a realizar una actividad guiada para probar a conectarnos a un servidor ECHO de nuestro ordenador desde un cliente en Android.

Lo primero es establecer en el ordenador un servidor ECHO, o encontrar una IP que tenga un servidor abierto.

Tras esto, crearemos una nueva aplicación y pondremos el siguiente código:

2. MODELO CLIENTE-SERVIDOR EN ANDROID

```
public class MainActivity extends Activity {
    private TextView output;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        output = (TextView) findViewById(R.id.TextView01);
        StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
                .permitNetwork().build());
        ejecutaCliente();
    }
    private void ejecutaCliente() {
        String ip = "192.168.2.1";
        int puerto = 7;
        log(" socket " + ip + " " + puerto);
        try {
            Socket sk = new Socket(ip, puerto);
            BufferedReader entrada = new BufferedReader(
                    new InputStreamReader(sk.getInputStream()));
            PrintWriter salida = new PrintWriter(
                    new OutputStreamWriter(sk.getOutputStream()), true);
            log("enviando... Hola Mundo ");
            salida.println("Hola Mundo");
            log("recibiendo ... " + entrada.readLine());
            sk.close();
        } catch (Exception e) {
            log("error: " + e.toString());
        }
    }
    private void log(String string) {
        output.append(string + "\n");
    }
}
```

2. MODELO CLIENTE-SERVIDOR EN ANDROID

Además, tenemos que solicitar en la aplicación permiso INTERNET en AndroidManifest.xml:

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

2. MODELO CLIENTE-SERVIDOR EN ANDROID

Si no hubiese un servidor de ECHO funcionando en el ordenador, se podría crear uno en Java con el siguiente código:

```
public class ServidorECHO {
   public static void main(String args[]) {
      try {
         System.out.println("Servidor en marcha...");
         ServerSocket sk = new ServerSocket(7);
         while (true) {
            Socket cliente = sk.accept();
            BufferedReader entrada = new BufferedReader(
                  new InputStreamReader(cliente.getInputStream()));
            PrintWriter salida = new PrintWriter(new OutputStreamWriter(
                  cliente.getOutputStream()), true);
            String datos = entrada.readLine();
            salida.println(datos);
            cliente.close();
            System.out.println(datos);
         }
      } catch (IOException e) {
         System.out.println(e);
      }
   }
}
```

1. GESTIÓN DE LA COMUNICACIÓN INALÁMBRICA

Además de la conectividad a la red por defecto que tiene Android, nos ofrece una APO que permite a nuestras aplicaciones interactuar y conectarse a otros dispositivos con protocolos como Bluetooth, NFC, Wi-Fi P2P y USB.

- **Bluetooth**: posibilita la transmisión de voz y datos entre diferentes dispositivos mediante un enlace por radiofrecuencia en la banda de los 2.4 GHz.
- **NFC**: comunicación de campo cercano. Es de corto alcance y alta frecuencia, y permite el intercambio de datos entre dispositivos.
- **Wi-Fi P2P**: uso del estándar IEEE 802.11 peer to peer.
- **USB**: permite comunicaciones mediante un cable llamado Bus Universal en Serie.

2. CONECTIVIDAD BLUETOOTH

La plataforma de Android incluye compatibilidad con la pila de red Bluetooth, la cual permite que un dispositivo intercambie datos de manera inalámbrica con otros dispositivos Bluetooth.

Con las Bluetooth API, una aplicación de Android puede realizar lo siguiente:

- Buscar otros dispositivos Bluetooth
- Consultar el adaptador local de Bluetooth en busca de dispositivos Bluetooth sincronizados
- Establecer canales RFCOMM
- Conectarse con otros dispositivos mediante el descubrimiento de servicios
- Transferir datos hacia otros dispositivos y desde estos
- Administrar varias conexiones

2. CONECTIVIDAD BLUETOOTH

Todas las Bluetooth API están disponibles en el paquete de android.bluetooth. Tenemos diferentes clases e interfaces que son útiles a la hora de crear conexiones Bluetooth:

BluetoothAdapter

Representa el adaptador local de Bluetooth (radio Bluetooth). El BluetoothAdapter es el punto de entrada de toda interacción de Bluetooth. Gracias a esto, puedes ver otros dispositivos Bluetooth, consultar una lista de los dispositivos conectados (sincronizados), crear una instancia de BluetoothDevice mediante una dirección MAC conocida y crear un BluetoothServerSocket para oír comunicaciones de otros dispositivos.

2. CONECTIVIDAD BLUETOOTH

BluetoothDevice

Representa un dispositivo Bluetooth remoto. Usa esto para solicitar una conexión con un dispositivo remoto mediante un BluetoothSocket o consultar información sobre el dispositivo, como su nombre, dirección, clase y estado de conexión.

BluetoothSocket

Representa la interfaz de un socket de Bluetooth (similar a un Socket de TCP). Este es el punto de conexión que permite que una aplicación intercambie datos con otro dispositivo Bluetooth a través de InputStream y OutputStream.

2. CONECTIVIDAD BLUETOOTH

BluetoothServerSocket

Representa un socket de servidor abierto que recibe solicitudes entrantes (similar a un ServerSocket de TCP). A fin de conectar dos dispositivos Android, un dispositivo debe abrir un socket de servidor con esta clase. Cuando un dispositivo Bluetooth remoto realice una solicitud de conexión a este dispositivo, el BluetoothServerSocket mostrará un BluetoothSocket conectado una vez que la conexión se acepte.

BluetoothClass

Describe las características y capacidades generales de un dispositivo Bluetooth. Se trata de un conjunto de propiedades de solo lectura que define las clases del dispositivo mayor y menor y sus servicios. Sin embargo, esto no describe de manera confiable todos los perfiles de Bluetooth y los servicios compatibles con el dispositivo, pero resulta útil como sugerencia para el tipo de dispositivo.

2. CONECTIVIDAD BLUETOOTH

Además, con el fin de utilizar las funciones de Bluetooth en la aplicación, se debe declarar el permiso BLUETOOTH en el Manifest.

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

2. CONECTIVIDAD BLUETOOTH

Los pasos para configurar el Bluetooth son:

1. Obtener el BluetoothAdapter

```
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
if (mBluetoothAdapter == null) {
    // Device does not support Bluetooth
}
```

2. Habilitar Bluetooth

```
if (!mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

3. CONSULTA DE DISPOSITIVOS SINCRONIZADOS

Antes de llevar a cabo la detección de dispositivos, es importante consultar el conjunto de dispositivos sincronizados a fin de ver si el dispositivo deseado ya es conocido. Para ello, llama a getBondedDevices(). Esto mostrará un conjunto de BluetoothDevice que representa los dispositivos sincronizados. Por ejemplo, puedes consultar todos los dispositivos sincronizados y luego mostrar el nombre de cada uno al usuario mediante ArrayAdapter:

```
Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();
// If there are paired devices
if (pairedDevices.size() > 0) {
    // Loop through paired devices
    for (BluetoothDevice device : pairedDevices) {
        // Add the name and address to an array adapter to show in a ListView
        mArrayAdapter.add(device.getName() + "\n" + device.getAddress());
    }
}
```

1. ENVÍO Y RECEPCIÓN DE MENSAJES DE TEXTO.

Se pueden construir un Intent para que enviemos el mensaje desde la aplicación original de SMS que tengamos instalada en el teléfono. Para ello, tendríamos que dar permiso android.permission.SEND_SMS en el AndroidManifest.xml y añadir la siguiente pieza de código a nuestra aplicación.

```
strPhone = "XXXXXXXXXXX";
strMessage = "LoremnIpsum";

Intent sendIntent = new Intent(Intent.ACTION_VIEW);
sendIntent.setType("vnd.android-dir/mms-sms");
sendIntent.putExtra("address", strPhone);
sendIntent.putExtra("sms_body", strMessage);
startActivity(sendIntent);
```

1. ENVÍO Y RECEPCIÓN DE MENSAJES DE TEXTO.

Otra opción que existe es ejecutar la url "smsto: + XXXXXXXXXX", que es procesada por Android, ejecutando la aplicación por defecto que esté asignada al protocolo smsto.

```
strPhone = "XXXXXXXXXX";
strMessage = "LoremnIpsum";

Uri sms_uri = Uri.parse("smsto:+" + strPhone);
Intent sms_intent = new Intent(Intent.ACTION_SENDTO, sms_uri);
sms_intent.putExtra("sms_body", txtMessage.getText().toString());
startActivity(sms_intent);
```

1. ENVÍO Y RECEPCIÓN DE MENSAJES DE TEXTO.

También se puede hacer sin utilizar un Intent, con la clase SmsManager. Ésta tiene un método sendTextMessage() que permitiría realizar dicha acción.

Un ejemplo sería:

```
String strPhone = "XXXXXXXXXX";

String strMessage = "LoremnIpsum";

SmsManager sms = SmsManager.getDefault();

sms.sendTextMessage(strPhone, null, strMessage, null, null);

Toast.makeText(this, "Sent.", Toast.LENGTH_SHORT).show();
```

ACTIVIDAD 6. Envío de SMS

Realiza una aplicación cumpla los siguientes requisitos:

- La vista ha de tener:
  - Una caja de texto en la que escribir hasta 160 caracteres.
  - Una línea de texto en la que añadir un número de teléfono.
  - Un botón de "enviar".
- En la parte de la lógica, debe poder enviar el texto introducido por SMS al teléfono introducido al darle al botón "enviar".

1. NETWORKING

Para realizar conexiones HTTP en Android, disponemos de la clase abstracta URLConnection, de las que salen HttpURLConnection y JarURLConnection.

En general, crear una conexión a una URL sigue los siguientes pasos:

1. Se establece un objeto conexión invocando opeConnection a una URL.
2. Se manipulan los parámetros generales y las peticiones.
3. Se establece una conexión con el objeto remoto, usando el método connect.
4. Una vez está disponible el objeto remoto, se puede acceder a sus métodos.

Sesión 9. Contenidos teóricos

1. NETWORKING

A la hora de implementar una conexión HTTP, tenemos que definir en el Manifest el permiso correspondiente para acceder a Internet:

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

1. Para crear la conexión con la URL, usaríamos HttpURLConnection (HttpsURLConnection para HTTPS).

```
URL url = new URL("http://url.com");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
//método http (GET, POST, PUT...)
connection.setRequestMethod("POST");
```

Sesión 9. Contenidos teóricos

1. NETWORKING

2. Definiríamos los temporizadores de conexión y lectura. Este proceso es opcional.

```
connection.setConnectTimeout(5000);
connection.setReadTimeout(10000);
```

3. Establecemos los parámetros del header, incluyendo autenticación BASIC si fuese necesario:

```
connection.setRequestProperty("Content-Type", "application/json");
connection.setRequestProperty("User-Agent","cliente Android 1.0");
//autenticación BASIC
connection.setRequestProperty("Authorization","Basic " + Base64.encodeToString((usuario
+ ":" + password).getBytes(), Base64.NO_WRAP));
```

1. NETWORKING

4. Incluimos el documento que se enviará en el body si es necesario, por ejemplo el JSON o XML de un servicio REST.

```
connection.setDoOutput(true);
DataOutputStream dataOutputStream = new
DataOutputStream(connection.getOutputStream());
dataOutputStream.write(json.getBytes(StandardCharsets.UTF_8));
dataOutputStream.flush();
dataOutputStream.close();
```

1. NETWORKING

5. Realizar la petición y comprobar el código de estado devuelto por el servidor web. La recomendación es usar las constantes públicas definidas en la clase HttpUrlConnection.

```
if(connection.getResponseCode() == HttpURLConnection.HTTP_OK)
```

6. Obtener la respuesta para procesarla

```
InputStream inputStreamResponse = connection.getInputStream();
```

Por ejemplo, si tenemos un documento de texto plano como respuesta, la convertimos en una cadena:

```
String linea = null;
StringBuilder respuestaCadena = new StringBuilder();
BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(inputStreamResponse, "UTF-8"));
while((linea = bufferedReader.readLine()) != null){
    respuestaCadena.append(linea);
}
```

1. NETWORKING

7. Cerrar el InputStream de respuesta en un finally. Si no hacemos el connection.disconnect(), el socket será liberado o reutilizado por el sistema.

```
if (inputStreamResponse != null){
  try{
    inputStreamResponse.close();
  }
  catch(IOException ex){
    Log.d(this.getClass().toString(), "Error cerrando InputStream", ex);
  }
}
```

2. COMPLEMENTOS PARA VISUALIZACIÓN DE SITIOS WEB MÓVILES

Los móviles son los dispositivos con los que más tiempo pasamos conectados a Internet, por lo que la visualización de sitios web móviles a la hora de realiza el diseño es muy importante.

Tenemos diferentes herramientas para este cometido:

- **Resizer**: herramienta de Google para comprobar el diseño en diferentes entornos. https://material.io/tools/resizer/
- **mobiReady**: nos permite ver cómo queda la web en diferentes tamaños de pantalla. https://ready.mobi/

3. PRUEBAS

Android Studio está diseñada para simplificar las pruebas, pudiendo establecer pruebas Junit con unos pocos clicks para que se ejecute en el JVM local, o una prueba instrumentada que se ejecute en algún dispositivo.

Además, también existen frameworks de prueba como Mockito, para llamadas de Android API de unidades locales, y Expresso o UI Automator, para ejercitar la interacción con usuarios en las pruebas instrumentadas.

3. PRUEBAS

Tipos de pruebas:

- **Pruebas de unidad local**.
  - Ubicación: module-name/src/test/java/.
  - Estas son pruebas que se ejecutan en la máquina virtual Java (JVM) local de la máquina. Se usan estas pruebas para minimizar el tiempo de ejecución cuando las pruebas no tengan dependencia de marco de Android o cuando se puedan simular las dependencia del marco de Android.

3. PRUEBAS

Tipos de pruebas:

- **Pruebas instrumentadas**.
  - Ubicación: module-name/src/androidTest/java/.
  - Estas son pruebas que se ejecutan en un dispositivo o emulador de hardware. Tienen acceso a las API Instrumentation, y permiten acceder a información como el Context de la app sobre la que se realizan pruebas, y controlar la app a prueba desde el código desarrollado para este cometido. Mayoritariamente son pruebas de IU funcionales e integradoras para automatizar la interacción de usuarios.

4. DOCUMENTACIÓN

Cuando se está desarrollando la aplicación con Java, se puede hacer uso de JavaDoc para la creación automática de documentación a partir de los comentarios del código.

Algunas de las etiquetas más importantes son:

- @author – Indica quién ha escrito dicha clase o método
- @version – Indica qué versión es
- @param – Indica los parámetros de dicho método
- @return – Indica el retorno de dicho método

Actividad 7. Creación de una prueba

Sigue los siguientes pasos para crear una prueba en una de las apps creadas.

Para crear una prueba de unidad local o una prueba instrumentada, puedes crear una prueba nueva para una clase o método específico siguiendo estos pasos:

1. Abre el archivo Java que contiene el código que desees probar.
2. Haz clic en la clase o método que desees probar, luego presiona Ctrl+Shift+T (⇧⌘T).
3. En el menú que aparece, haz clic en Create New Test.
4. En el diálogo Create Test, edita cualquier campo, selecciona cualquier método que generarás y luego haz clic en OK.
5. En el diálogo Choose Destination Directory, haz clic en el conjunto de orígenes correspondiente al tipo de prueba que desees crear: androidTest para una prueba instrumentada o test para una prueba de unidad local. Luego haz clic en OK.