GUÍA DE ESTUDIO: PROGRAMACIÓN MULTIMEDIA DE DISPOSITIVOS MÓVILES

Índice de Contenidos

Introducción

Esta guía ha sido elaborada para ayudarte a preparar el examen de junio de la asignatura de Programación Multimedia de Dispositivos Móviles. Está basada en el material teórico proporcionado y en el análisis de exámenes anteriores, con el objetivo de ofrecerte una herramienta completa para abordar con éxito la evaluación.

La asignatura se centra en el desarrollo de aplicaciones para dispositivos móviles, principalmente en la plataforma Android, abarcando desde los conceptos fundamentales hasta aspectos más avanzados como la persistencia de datos, la conectividad y los servicios REST.

A lo largo de esta guía, encontrarás explicaciones claras y concisas de cada tema, acompañadas de ejemplos prácticos y consejos específicos para resolver los tipos de ejercicios que suelen aparecer en los exámenes. También se incluye un resumen de

conceptos clave que te será útil para repasar rápidamente los puntos más importantes antes del examen.

Tecnologías de desarrollo para dispositivos móviles

Características de los dispositivos móviles

Los dispositivos móviles se caracterizan por:

- **Movilidad**: Permiten su uso en cualquier lugar.
- **Conectividad**: Disponen de diversas tecnologías de conexión (WiFi, Bluetooth, NFC, etc.).
- **Capacidades limitadas**: Aunque cada vez más potentes, siguen teniendo limitaciones en comparación con equipos de escritorio.
- **Interfaz táctil**: Principal método de interacción.
- **Sensores**: Acelerómetro, giroscopio, GPS, etc., que permiten nuevas formas de interacción.

Plataformas móviles

Las principales plataformas móviles son:

- **Android**: Sistema operativo de Google, basado en Linux. Es el más extendido a nivel mundial.
- **iOS**: Sistema operativo de Apple para iPhone y iPad.
- **Otros**: Windows Phone (descontinuado), KaiOS, etc.

Arquitectura de Android

Android es un sistema operativo basado en Linux, con una arquitectura en capas:

1. **Kernel de Linux**: Base del sistema operativo, gestiona procesos, memoria, etc.
2. **Capa de abstracción de hardware (HAL)**: Proporciona interfaces estándar para los componentes hardware.
3. **Bibliotecas nativas**: Incluyen SQLite, WebKit, OpenGL, etc.
4. **Runtime de Android**: Dalvik VM (hasta Android 4.4) o ART (Android Runtime, desde Android 5.0).
5. **Framework de aplicaciones**: APIs que utilizan las aplicaciones.
6. **Aplicaciones**: Nivel superior, donde se encuentran las apps instaladas.

Componentes de una aplicación Android

Los principales componentes son:

- **Activities**: Representan una pantalla con interfaz de usuario.
- **Services**: Componentes que se ejecutan en segundo plano.
- **Broadcast Receivers**: Responden a eventos del sistema.
- **Content Providers**: Gestionan el acceso a datos compartidos.

Entorno de desarrollo

El entorno de desarrollo oficial para Android es **Android Studio**, basado en IntelliJ IDEA. Proporciona:

- Editor de código
- Herramientas de depuración
- Emulador de dispositivos
- Herramientas de análisis de rendimiento
- Sistema de construcción basado en Gradle

Estructura de un proyecto Android

Un proyecto Android en Android Studio tiene la siguiente estructura:

- **app/**: Contiene el código fuente y recursos de la aplicación.
- **src/main/java/**: Código fuente Java o Kotlin.
- **src/main/res/**: Recursos (layouts, imágenes, strings, etc.).
- **src/main/AndroidManifest.xml**: Configuración de la aplicación.
- **gradle/**: Scripts de configuración de Gradle.
- **build.gradle**: Archivos de configuración de la compilación.

Diseño de interfaces

Layouts

Los layouts en Android definen la estructura visual de la interfaz de usuario. Los principales tipos son:

LinearLayout

Organiza los elementos en una única dirección, horizontal o vertical.

```
<LinearLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">

  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Primer elemento" />

  <Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Segundo elemento" />
</LinearLayout>
```

Atributos importantes: - `android:orientation`: "vertical" u "horizontal" - `android:gravity`: Alineación de los elementos dentro del layout - `android:layout_weight`: Distribución proporcional del espacio

RelativeLayout

Permite posicionar los elementos en relación a otros elementos o al contenedor.

```
<RelativeLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:id="@+id/centerButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:text="Centro" />

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_above="@id/centerButton"
    android:layout_centerHorizontal="true"
    android:text="Encima del botón" />
</RelativeLayout>
```

Atributos importantes: - `android:layout_centerInParent`, `android:layout_centerHorizontal`, `android:layout_centerVertical` - `android:layout_above`, `android:layout_below`, `android:layout_toLeftOf`,

`android:layout_toRightOf` - `android:layout_alignParentTop`, `android:layout_alignParentBottom`, etc.

ConstraintLayout

Es el layout más flexible y recomendado actualmente. Permite crear diseños complejos con una jerarquía plana.

```
<androidx.constraintlayout.widget.ConstraintLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Botón"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Texto"
    app:layout_constraintBottom_toTopOf="@+id/button"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Atributos importantes: - `app:layout_constraint[Start|End|Top|Bottom]_to[Start|End|Top|Bottom]Of` - `app:layout_constraintHorizontal_bias`, `app:layout_constraintVertical_bias` - `app:layout_constraintDimensionRatio`

FrameLayout

Diseñado para mostrar un único elemento o para superponer elementos.

```
<FrameLayout
  android:layout_width="match_parent"
  android:layout_height="200dp">

  <ImageView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scaleType="centerCrop"
    android:src="@drawable/background" />
```

```
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Texto superpuesto"
        android:textColor="#FFFFFF" />
</FrameLayout>
```

Atributos importantes: - `android:layout_gravity` : Posición del elemento dentro del FrameLayout

TableLayout

Organiza los elementos en filas y columnas.

```
<TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TableRow>
        <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Columna 1" />

        <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Columna 2" />
    </TableRow>

    <TableRow>
        <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Dato 1" />

        <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Dato 2" />
    </TableRow>
</TableLayout>
```

Atributos importantes: - `android:stretchColumns` , `android:shrinkColumns` , `android:collapseColumns`

Controles básicos

Los controles básicos son los elementos de interfaz de usuario más comunes en Android.

TextView

Muestra texto no editable.

```
<TextView
  android:id="@+id/textView"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Hola Mundo"
  android:textSize="18sp"
  android:textColor="#000000"
  android:textStyle="bold" />
```

Atributos importantes: - `android:text` : Texto a mostrar - `android:textSize` : Tamaño del texto - `android:textColor` : Color del texto - `android:textStyle` : Estilo del texto (normal, bold, italic) - `android:gravity` : Alineación del texto dentro del TextView

EditText

Permite al usuario introducir y editar texto.

```
<EditText
  android:id="@+id/editText"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:hint="Introduce tu nombre"
  android:inputType="text" />
```

Atributos importantes: - `android:hint` : Texto de sugerencia cuando está vacío - `android:inputType` : Tipo de entrada (text, textPassword, number, phone, etc.) - `android:maxLines` : Número máximo de líneas - `android:imeOptions` : Opciones para el teclado virtual (actionDone, actionNext, etc.)

Button

Elemento interactivo que el usuario puede pulsar para iniciar una acción.

```
<Button
  android:id="@+id/button"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Pulsar"
  android:onClick="onButtonClick" />
```

Atributos importantes: - `android:text` : Texto del botón - `android:onClick` : Método que se ejecutará al pulsar el botón

ImageButton

Similar a Button, pero muestra una imagen en lugar de texto.

```
<ImageButton
  android:id="@+id/imageButton"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:src="@drawable/ic_button"
  android:contentDescription="Botón de imagen" />
```

Atributos importantes: - `android:src` : Imagen a mostrar - `android:contentDescription` : Descripción para accesibilidad

ImageView

Muestra imágenes.

```
<ImageView
  android:id="@+id/imageView"
  android:layout_width="200dp"
  android:layout_height="200dp"
  android:src="@drawable/imagen"
  android:scaleType="centerCrop" />
```

Atributos importantes: - `android:src` : Imagen a mostrar - `android:scaleType` : Cómo escalar la imagen (centerCrop, fitCenter, etc.)

CheckBox

Permite seleccionar múltiples opciones.

```
<CheckBox
  android:id="@+id/checkBox"
  android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:text="Acepto los términos y condiciones" />
```

Atributos importantes: - `android:text` : Texto asociado al CheckBox - `android:checked` : Estado inicial (true/false)

RadioButton y RadioGroup

Permite seleccionar una única opción de un grupo.

```
<RadioGroup
    android:id="@+id/radioGroup"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <RadioButton
        android:id="@+id/radioButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Opción 1" />

    <RadioButton
        android:id="@+id/radioButton2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Opción 2" />

    <RadioButton
        android:id="@+id/radioButton3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Opción 3" />
</RadioGroup>
```

Atributos importantes: - `android:orientation` : Orientación del RadioGroup (vertical u horizontal) - `android:checked` : Estado inicial de cada RadioButton (true/false)

Spinner

Desplegable que permite seleccionar un elemento de una lista.

```
<Spinner
    android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:prompt="@string/spinner_prompt" />
```

En el código Java:

```
 Spinner spinner = findViewById(R.id.spinner);
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
        R.array.planets_array, android.R.layout.simple_spinner_item);
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
spinner.setAdapter(adapter);
```

ProgressBar

Muestra el progreso de una operación.

```
<ProgressBar
  android:id="@+id/progressBar"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  style="?android:attr/progressBarStyleHorizontal"
  android:max="100"
  android:progress="50" />
```

Atributos importantes: - `style` : Estilo de la barra de progreso (circular o horizontal) - `android:max` : Valor máximo - `android:progress` : Valor actual

SeekBar

Permite seleccionar un valor dentro de un rango.

```
<SeekBar
  android:id="@+id/seekBar"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:max="100"
  android:progress="50" />
```

Atributos importantes: - `android:max` : Valor máximo - `android:progress` : Valor inicial

Actividades e intents

Actividades (Activities)

Una actividad representa una pantalla con interfaz de usuario. Cada actividad es una clase que extiende de `Activity` o `AppCompatActivity` .

Ciclo de vida de una actividad:

1. **onCreate()**: Se llama cuando se crea la actividad. Aquí se inicializan los componentes y se establece el layout.
2. **onStart()**: Se llama cuando la actividad se hace visible para el usuario.
3. **onResume()**: Se llama cuando la actividad comienza a interactuar con el usuario.
4. **onPause()**: Se llama cuando la actividad pierde el foco pero sigue siendo visible.
5. **onStop()**: Se llama cuando la actividad ya no es visible para el usuario.
6. **onDestroy()**: Se llama antes de que la actividad sea destruida.
7. **onRestart()**: Se llama cuando la actividad vuelve a ser visible después de haber sido detenida.

Ejemplo de una actividad básica:

```
public class MainActivity extends AppCompatActivity {

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

        // Inicialización de componentes
        Button button = findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Acción al pulsar el botón
            }
        });
    }

    @Override
    protected void onResume() {
        super.onResume();
        // Código a ejecutar cuando la actividad se reanuda
    }

    @Override
    protected void onPause() {
        super.onPause();
        // Código a ejecutar cuando la actividad se pausa
    }
}
```

Intents

Los intents son objetos de mensajería que se utilizan para solicitar una acción de otro componente de la aplicación. Se utilizan principalmente para:

1. Iniciar una actividad
2. Iniciar un servicio
3. Entregar un broadcast

Tipos de intents:

- **Intents explícitos**: Especifican el componente exacto a iniciar.
- **Intents implícitos**: Solicitan una acción sin especificar el componente.

Ejemplo de intent explícito:

```
// Iniciar una nueva actividad
Intent intent = new Intent(MainActivity.this, SecondActivity.class);
startActivity(intent);
```

Ejemplo de intent implícito:

```
// Abrir una URL en el navegador
Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("https://
www.example.com"));
startActivity(intent);
```

Paso de datos entre actividades

Se pueden pasar datos entre actividades utilizando extras en los intents:

```
// En la actividad de origen
Intent intent = new Intent(MainActivity.this, SecondActivity.class);
intent.putExtra("nombre", "Juan");
intent.putExtra("edad", 25);
startActivity(intent);

// En la actividad de destino
String nombre = getIntent().getStringExtra("nombre");
int edad = getIntent().getIntExtra("edad", 0); // 0 es el valor por defecto
```

Para objetos complejos, estos deben implementar `Serializable` o `Parcelable`:

```
// Clase Persona implementando Serializable
public class Persona implements Serializable {
```

```
    private String nombre;
    private int edad;

    // Constructor, getters y setters
}

// Pasar un objeto Persona
Persona persona = new Persona("Juan", 25);
Intent intent = new Intent(MainActivity.this, SecondActivity.class);
intent.putExtra("persona", persona);
startActivity(intent);

// Recibir el objeto Persona
Persona persona = (Persona) getIntent().getSerializableExtra("persona");
```

Recibir resultados de actividades

Para recibir un resultado de una actividad, se utiliza `startActivityForResult()`:

```
// Iniciar actividad esperando resultado
Intent intent = new Intent(MainActivity.this, FormActivity.class);
startActivityForResult(intent, REQUEST_CODE);

// En la actividad de destino, establecer el resultado
Intent resultIntent = new Intent();
resultIntent.putExtra("resultado", "Datos procesados");
setResult(RESULT_OK, resultIntent);
finish();

// En la actividad de origen, recibir el resultado
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == REQUEST_CODE && resultCode == RESULT_OK) {
        String resultado = data.getStringExtra("resultado");
        // Procesar el resultado
    }
}
```

Controles avanzados

ListView y RecyclerView

Estos controles permiten mostrar listas de elementos con scroll.

ListView

```
<ListView
    android:id="@+id/listView"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

En el código Java:

```
ListView listView = findViewById(R.id.listView);
String[] items = {"Item 1", "Item 2", "Item 3", "Item 4", "Item 5"};
ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_list_item_1, items);
listView.setAdapter(adapter);

listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        String item = (String) parent.getItemAtPosition(position);
        Toast.makeText(MainActivity.this, "Seleccionado: " + item,
Toast.LENGTH_SHORT).show();
    }
});
```

RecyclerView

RecyclerView es más flexible y eficiente que ListView, y es el recomendado actualmente.

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

En el código Java:

```
// Definir el adaptador
public class MyAdapter extends RecyclerView.Adapter<MyAdapter.ViewHolder> {
    private String[] data;

    public MyAdapter(String[] data) {
        this.data = data;
    }

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext())
                .inflate(android.R.layout.simple_list_item_1, parent, false);
```

```
        return new ViewHolder(view);
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position) {
        holder.textView.setText(data[position]);
    }

    @Override
    public int getItemCount() {
        return data.length;
    }

    public static class ViewHolder extends RecyclerView.ViewHolder {
        public TextView textView;

        public ViewHolder(View view) {
            super(view);
            textView = view.findViewById(android.R.id.text1);
        }
    }
}

// Configurar el RecyclerView
RecyclerView recyclerView = findViewById(R.id.recyclerView);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
String[] data = {"Item 1", "Item 2", "Item 3", "Item 4", "Item 5"};
MyAdapter adapter = new MyAdapter(data);
recyclerView.setAdapter(adapter);
```

Adaptadores

Los adaptadores son el puente entre los datos y las vistas que los muestran. Los principales tipos son:

- **ArrayAdapter**: Para datos simples como arrays o listas.
- **SimpleAdapter**: Para mapas de datos.
- **CursorAdapter**: Para datos de una base de datos (Cursor).
- **BaseAdapter**: Adaptador personalizado para cualquier tipo de datos.
- **RecyclerView.Adapter**: Para RecyclerView.

Ejemplo de ArrayAdapter personalizado:

```
public class PersonaAdapter extends ArrayAdapter<Persona> {
    private Context context;
    private List<Persona> personas;

    public PersonaAdapter(Context context, List<Persona> personas) {
        super(context, 0, personas);
```

```
        this.context = context;
        this.personas = personas;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        // Obtener el objeto Persona para esta posición
        Persona persona = getItem(position);

        // Comprobar si se está reutilizando una vista, si no, inflarla
        if (convertView == null) {
            convertView =
LayoutInflater.from(getContext()).inflate(R.layout.item_persona, parent, false);
        }

        // Obtener referencias a las vistas
        TextView textViewNombre = convertView.findViewById(R.id.textViewNombre);
        TextView textViewEdad = convertView.findViewById(R.id.textViewEdad);

        // Establecer los datos en las vistas
        textViewNombre.setText(persona.getNombre());
        textViewEdad.setText(String.valueOf(persona.getEdad()));

        return convertView;
    }
}
```

Menús

Android soporta diferentes tipos de menús:

Options Menu (Menú de opciones)

Aparece en la barra de acción (ActionBar).

```
 @Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();

    if (id == R.id.action_settings) {
        // Acción para el ítem "Settings"
        return true;
    }
```

```
    return super.onOptionsItemSelected(item);
}
```

Archivo de menú (res/menu/menu_main.xml):

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/action_settings"
        android:title="Settings"
        android:orderInCategory="100"
        app:showAsAction="never" />

    <item
        android:id="@+id/action_search"
        android:title="Search"
        android:icon="@drawable/ic_search"
        app:showAsAction="ifRoom" />
</menu>
```

Context Menu (Menú contextual)

Aparece al mantener pulsado un elemento.

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenu.ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    getMenuInflater().inflate(R.menu.menu_context, menu);
}

@Override
public boolean onContextItemSelected(MenuItem item) {
    AdapterView.AdapterContextMenuInfo info =
(AdapterView.AdapterContextMenuInfo) item.getMenuInfo();
    int position = info.position;

    switch (item.getItemId()) {
        case R.id.action_edit:
            // Editar el elemento en la posición 'position'
            return true;
        case R.id.action_delete:
            // Eliminar el elemento en la posición 'position'
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

```
// Registrar una vista para el menú contextual
registerForContextMenu(listView);
```

Popup Menu

Menú emergente anclado a una vista.

```
 Button button = findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        PopupMenu popup = new PopupMenu(MainActivity.this, v);
        popup.getMenuInflater().inflate(R.menu.menu_popup, popup.getMenu());

        popup.setOnMenuItemClickListener(new
PopupMenu.OnMenuItemClickListener() {
            @Override
            public boolean onMenuItemClick(MenuItem item) {
                switch (item.getItemId()) {
                    case R.id.action_item1:
                        // Acción para el ítem 1
                        return true;
                    case R.id.action_item2:
                        // Acción para el ítem 2
                        return true;
                    default:
                        return false;
                }
            }
        });

        popup.show();
    }
});
```

Diálogos

Los diálogos son ventanas pequeñas que solicitan una decisión o información adicional al usuario.

AlertDialog

```
 AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Título del diálogo")
       .setMessage("¿Estás seguro de que quieres realizar esta acción?")
       .setPositiveButton("Sí", new DialogInterface.OnClickListener() {
           @Override
           public void onClick(DialogInterface dialog, int which) {
```

```
            // Acción al pulsar "Sí"
        }
    })
    .setNegativeButton("No", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            // Acción al pulsar "No"
            dialog.dismiss();
        }
    })
    .show();
```

DatePickerDialog

```
 Calendar calendar = Calendar.getInstance();
int year = calendar.get(Calendar.YEAR);
int month = calendar.get(Calendar.MONTH);
int day = calendar.get(Calendar.DAY_OF_MONTH);

DatePickerDialog datePickerDialog = new DatePickerDialog(this,
        new DatePickerDialog.OnDateSetListener() {
            @Override
            public void onDateSet(DatePicker view, int year, int month, int dayOfMonth)
{
                // Procesar la fecha seleccionada
                String fecha = dayOfMonth + "/" + (month + 1) + "/" + year;
                Toast.makeText(MainActivity.this, "Fecha: " + fecha,
Toast.LENGTH_SHORT).show();
            }
        }, year, month, day);
datePickerDialog.show();
```

TimePickerDialog

```
 Calendar calendar = Calendar.getInstance();
int hour = calendar.get(Calendar.HOUR_OF_DAY);
int minute = calendar.get(Calendar.MINUTE);

TimePickerDialog timePickerDialog = new TimePickerDialog(this,
        new TimePickerDialog.OnTimeSetListener() {
            @Override
            public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
                // Procesar la hora seleccionada
                String hora = hourOfDay + ":" + minute;
                Toast.makeText(MainActivity.this, "Hora: " + hora,
Toast.LENGTH_SHORT).show();
            }
        }, hour, minute, true); // true para formato 24h
timePickerDialog.show();
```

Notificaciones

Las notificaciones informan al usuario sobre eventos importantes, incluso cuando la aplicación no está en primer plano.

```
 private void createNotification() {
    // Crear un canal de notificación (requerido para Android 8.0+)
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        String channelId = "mi_canal_id";
        CharSequence channelName = "Mi Canal";
        int importance = NotificationManager.IMPORTANCE_DEFAULT;
        NotificationChannel channel = new NotificationChannel(channelId,
channelName, importance);

        NotificationManager notificationManager =
getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }

    // Crear un intent para cuando se pulse la notificación
    Intent intent = new Intent(this, MainActivity.class);
    PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent, 0);

    // Construir la notificación
    NotificationCompat.Builder builder = new NotificationCompat.Builder(this,
"mi_canal_id")
            .setSmallIcon(R.drawable.ic_notification)
            .setContentTitle("Título de la notificación")
            .setContentText("Texto de la notificación")
            .setPriority(NotificationCompat.PRIORITY_DEFAULT)
            .setContentIntent(pendingIntent)
            .setAutoCancel(true);

    // Mostrar la notificación
    NotificationManagerCompat notificationManager =
NotificationManagerCompat.from(this);
    notificationManager.notify(1, builder.build());
}
```

Fragmentos

Los fragmentos son componentes modulares de la interfaz de usuario que tienen su propio ciclo de vida y pueden ser reutilizados en diferentes actividades.

Ciclo de vida de un fragmento

El ciclo de vida de un fragmento es similar al de una actividad, pero con algunos métodos adicionales:

1. **onAttach()**: Se llama cuando el fragmento se asocia a una actividad.
2. **onCreate()**: Se llama cuando se crea el fragmento.
3. **onCreateView()**: Se llama para crear la vista del fragmento.
4. **onViewCreated()**: Se llama después de que la vista del fragmento ha sido creada.
5. **onStart()**: Se llama cuando el fragmento se hace visible.
6. **onResume()**: Se llama cuando el fragmento comienza a interactuar con el usuario.
7. **onPause()**: Se llama cuando el fragmento pierde el foco.
8. **onStop()**: Se llama cuando el fragmento ya no es visible.
9. **onDestroyView()**: Se llama cuando la vista del fragmento está siendo destruida.
10. **onDestroy()**: Se llama cuando el fragmento está siendo destruido.
11. **onDetach()**: Se llama cuando el fragmento se desasocia de la actividad.

Creación de un fragmento

```
public class MiFragmento extends Fragment {

    public MiFragmento() {
        // Constructor vacío requerido
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
        // Inflar el layout para este fragmento
        return inflater.inflate(R.layout.fragment_mi_fragmento, container, false);
    }

    @Override
    public void onViewCreated(@NonNull View view, @Nullable Bundle
savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);

        // Inicializar componentes de la vista
        Button button = view.findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Acción al pulsar el botón
            }
        });
```

```
    }
}
```

Layout del fragmento (res/layout/fragment_mi_fragmento.xml):

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Este es mi fragmento" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Pulsar" />
</LinearLayout>
```

Añadir un fragmento a una actividad

Hay dos formas de añadir un fragmento a una actividad:

1. Declarativamente (en el XML)

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/fragmentContainerView"
    android:name="com.example.app.MiFragmento"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

2. Programáticamente (en el código Java)

```
MiFragmento fragmento = new MiFragmento();
getSupportFragmentManager().beginTransaction()
        .add(R.id.fragmentContainer, fragmento)
        .commit();
```

Transacciones de fragmentos

Las transacciones permiten añadir, reemplazar, ocultar o mostrar fragmentos dinámicamente.

```
// Reemplazar un fragmento
FragmentoB fragmentoB = new FragmentoB();
getSupportFragmentManager().beginTransaction()
        .replace(R.id.fragmentContainer, fragmentoB)
        .commit();

// Añadir un fragmento a la pila de retroceso
FragmentoC fragmentoC = new FragmentoC();
getSupportFragmentManager().beginTransaction()
        .replace(R.id.fragmentContainer, fragmentoC)
        .addToBackStack(null) // Permite volver al fragmento anterior con el botón Atrás
        .commit();

// Ocultar y mostrar fragmentos
getSupportFragmentManager().beginTransaction()
        .hide(fragmentoA)
        .show(fragmentoB)
        .commit();
```

Comunicación entre fragmentos y actividades

La comunicación entre fragmentos debe realizarse a través de la actividad que los contiene, utilizando interfaces.

```
// En el fragmento
public interface OnFragmentInteractionListener {
    void onFragmentInteraction(String data);
}

private OnFragmentInteractionListener mListener;

@Override
public void onAttach(Context context) {
    super.onAttach(context);
    if (context instanceof OnFragmentInteractionListener) {
        mListener = (OnFragmentInteractionListener) context;
    } else {
        throw new RuntimeException(context.toString() + " debe implementar
OnFragmentInteractionListener");
    }
}

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

// Método para enviar datos a la actividad
```

```
private void enviarDatos(String data) {
    if (mListener != null) {
        mListener.onFragmentInteraction(data);
    }
}

// En la actividad
public class MainActivity extends AppCompatActivity implements
MiFragmento.OnFragmentInteractionListener {

    @Override
    public void onFragmentInteraction(String data) {
        // Procesar los datos recibidos del fragmento
        // O pasarlos a otro fragmento
        OtroFragmento otroFragmento = (OtroFragmento)
getSupportFragmentManager().findFragmentById(R.id.otroFragmento);
        if (otroFragmento != null) {
            otroFragmento.actualizarDatos(data);
        }
    }
}
```

Persistencia de datos

La persistencia de datos permite almacenar información de forma permanente, para que esté disponible incluso después de cerrar la aplicación.

Preferencias compartidas (SharedPreferences)

Las preferencias compartidas son adecuadas para almacenar pequeñas cantidades de datos primitivos en forma de pares clave-valor.

```
 // Guardar datos
SharedPreferences sharedPreferences = getSharedPreferences("MisPreferencias",
MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPreferences.edit();
editor.putString("nombre", "Juan");
editor.putInt("edad", 25);
editor.putBoolean("activo", true);
editor.apply(); // o editor.commit() para guardar de forma síncrona

// Leer datos
SharedPreferences sharedPreferences = getSharedPreferences("MisPreferencias",
MODE_PRIVATE);
String nombre = sharedPreferences.getString("nombre", ""); // "" es el valor por
defecto
```

```
int edad = sharedPreferences.getInt("edad", 0);
boolean activo = sharedPreferences.getBoolean("activo", false);
```

Almacenamiento de archivos

Android proporciona varias opciones para almacenar archivos:

Almacenamiento interno

Los archivos se guardan en el directorio privado de la aplicación y se eliminan cuando se desinstala la aplicación.

```
// Escribir en un archivo
try {
    FileOutputStream fos = openFileOutput("archivo.txt", Context.MODE_PRIVATE);
    fos.write("Contenido del archivo".getBytes());
    fos.close();
} catch (IOException e) {
    e.printStackTrace();
}

// Leer de un archivo
try {
    FileInputStream fis = openFileInput("archivo.txt");
    InputStreamReader isr = new InputStreamReader(fis);
    BufferedReader br = new BufferedReader(isr);
    StringBuilder sb = new StringBuilder();
    String line;
    while ((line = br.readLine()) != null) {
        sb.append(line);
    }
    fis.close();
    String contenido = sb.toString();
} catch (IOException e) {
    e.printStackTrace();
}
```

Almacenamiento externo

Los archivos se guardan en el almacenamiento externo (tarjeta SD o almacenamiento interno compartido) y pueden ser accesibles por otras aplicaciones.

```
// Comprobar si el almacenamiento externo está disponible
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    return Environment.MEDIA_MOUNTED.equals(state);
}
```

```java
// Obtener el directorio de archivos externos de la aplicación
File file = new File(getExternalFilesDir(null), "archivo.txt");

// Escribir en un archivo
try {
    FileOutputStream fos = new FileOutputStream(file);
    fos.write("Contenido del archivo".getBytes());
    fos.close();
} catch (IOException e) {
    e.printStackTrace();
}

// Leer de un archivo
try {
    FileInputStream fis = new FileInputStream(file);
    InputStreamReader isr = new InputStreamReader(fis);
    BufferedReader br = new BufferedReader(isr);
    StringBuilder sb = new StringBuilder();
    String line;
    while ((line = br.readLine()) != null) {
        sb.append(line);
    }
    fis.close();
    String contenido = sb.toString();
} catch (IOException e) {
    e.printStackTrace();
}
```

Base de datos SQLite

SQLite es una base de datos relacional ligera que viene integrada en Android.

Definición del esquema

```java
public class ContratoBaseDatos {
    // Para evitar que alguien instancie esta clase
    private ContratoBaseDatos() {}

    // Definición de la tabla
    public static class TablaUsuarios implements BaseColumns {
        public static final String NOMBRE_TABLA = "usuarios";
        public static final String COLUMNA_NOMBRE = "nombre";
        public static final String COLUMNA_EMAIL = "email";
        public static final String COLUMNA_EDAD = "edad";
    }

    // Sentencias SQL
    public static final String SQL_CREAR_TABLA =
            "CREATE TABLE " + TablaUsuarios.NOMBRE_TABLA + " (" +
```

```
            TablaUsuarios._ID + " INTEGER PRIMARY KEY," +
            TablaUsuarios.COLUMNA_NOMBRE + " TEXT," +
            TablaUsuarios.COLUMNA_EMAIL + " TEXT," +
            TablaUsuarios.COLUMNA_EDAD + " INTEGER)";

    public static final String SQL_ELIMINAR_TABLA =
            "DROP TABLE IF EXISTS " + TablaUsuarios.NOMBRE_TABLA;
}
```

Helper de la base de datos

```
public class MiDBHelper extends SQLiteOpenHelper {
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "MiBaseDatos.db";

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

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(ContratoBaseDatos.SQL_CREAR_TABLA);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // Política simple: eliminar y recrear la tabla
        db.execSQL(ContratoBaseDatos.SQL_ELIMINAR_TABLA);
        onCreate(db);
    }
}
```

Operaciones CRUD (Create, Read, Update, Delete)

```
public class UsuarioDAO {
    private MiDBHelper dbHelper;

    public UsuarioDAO(Context context) {
        dbHelper = new MiDBHelper(context);
    }

    // Insertar un usuario
    public long insertarUsuario(String nombre, String email, int edad) {
        SQLiteDatabase db = dbHelper.getWritableDatabase();

        ContentValues values = new ContentValues();
        values.put(ContratoBaseDatos.TablaUsuarios.COLUMNA_NOMBRE, nombre);
        values.put(ContratoBaseDatos.TablaUsuarios.COLUMNA_EMAIL, email);
        values.put(ContratoBaseDatos.TablaUsuarios.COLUMNA_EDAD, edad);
```

```
        return db.insert(ContratoBaseDatos.TablaUsuarios.NOMBRE_TABLA, null,
values);
    }

    // Consultar todos los usuarios
    public Cursor obtenerTodosLosUsuarios() {
        SQLiteDatabase db = dbHelper.getReadableDatabase();

        String[] projection = {
                ContratoBaseDatos.TablaUsuarios._ID,
                ContratoBaseDatos.TablaUsuarios.COLUMNA_NOMBRE,
                ContratoBaseDatos.TablaUsuarios.COLUMNA_EMAIL,
                ContratoBaseDatos.TablaUsuarios.COLUMNA_EDAD
        };

        return db.query(
                ContratoBaseDatos.TablaUsuarios.NOMBRE_TABLA,
                projection,
                null,
                null,
                null,
                null,
                null
        );
    }

    // Actualizar un usuario
    public int actualizarUsuario(long id, String nombre, String email, int edad) {
        SQLiteDatabase db = dbHelper.getWritableDatabase();

        ContentValues values = new ContentValues();
        values.put(ContratoBaseDatos.TablaUsuarios.COLUMNA_NOMBRE, nombre);
        values.put(ContratoBaseDatos.TablaUsuarios.COLUMNA_EMAIL, email);
        values.put(ContratoBaseDatos.TablaUsuarios.COLUMNA_EDAD, edad);

        String selection = ContratoBaseDatos.TablaUsuarios._ID + " = ?";
        String[] selectionArgs = { String.valueOf(id) };

        return db.update(
                ContratoBaseDatos.TablaUsuarios.NOMBRE_TABLA,
                values,
                selection,
                selectionArgs
        );
    }

    // Eliminar un usuario
    public int eliminarUsuario(long id) {
        SQLiteDatabase db = dbHelper.getWritableDatabase();

        String selection = ContratoBaseDatos.TablaUsuarios._ID + " = ?";
```

```
        String[] selectionArgs = { String.valueOf(id) };

        return db.delete(
                ContratoBaseDatos.TablaUsuarios.NOMBRE_TABLA,
                selection,
                selectionArgs
        );
    }

    // Cerrar la conexión
    public void cerrar() {
        dbHelper.close();
    }
}
```

Uso del DAO

```
 UsuarioDAO usuarioDAO = new UsuarioDAO(this);

// Insertar un usuario
long id = usuarioDAO.insertarUsuario("Juan", "juan@example.com", 25);

// Consultar todos los usuarios
Cursor cursor = usuarioDAO.obtenerTodosLosUsuarios();
while (cursor.moveToNext()) {
    long itemId =
cursor.getLong(cursor.getColumnIndexOrThrow(ContratoBaseDatos.TablaUsuarios._ID));
    String nombre =
cursor.getString(cursor.getColumnIndexOrThrow(ContratoBaseDatos.TablaUsuarios.COLUMNA
    String email =
cursor.getString(cursor.getColumnIndexOrThrow(ContratoBaseDatos.TablaUsuarios.COLUMNA
    int edad =
cursor.getInt(cursor.getColumnIndexOrThrow(ContratoBaseDatos.TablaUsuarios.COLUMNA_E

    // Hacer algo con los datos
}
cursor.close();

// Actualizar un usuario
int filasActualizadas = usuarioDAO.actualizarUsuario(id, "Juan Pérez",
"juan.perez@example.com", 26);

// Eliminar un usuario
int filasEliminadas = usuarioDAO.eliminarUsuario(id);

// Cerrar la conexión
usuarioDAO.cerrar();
```

Room (Biblioteca de persistencia)

Room es una capa de abstracción sobre SQLite que facilita el acceso a la base de datos.

Entidad

```
@Entity(tableName = "usuarios")
public class Usuario {
    @PrimaryKey(autoGenerate = true)
    private int id;

    @ColumnInfo(name = "nombre")
    private String nombre;

    @ColumnInfo(name = "email")
    private String email;

    @ColumnInfo(name = "edad")
    private int edad;

    // Constructor, getters y setters

    public Usuario(String nombre, String email, int edad) {
        this.nombre = nombre;
        this.email = email;
        this.edad = edad;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

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

    public String getEmail() {
        return email;
    }

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

```
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}
```

DAO (Data Access Object)

```
@Dao
public interface UsuarioDao {
    @Insert
    long insert(Usuario usuario);

    @Query("SELECT * FROM usuarios")
    List<Usuario> getAll();

    @Query("SELECT * FROM usuarios WHERE id = :id")
    Usuario getById(int id);

    @Update
    int update(Usuario usuario);

    @Delete
    int delete(Usuario usuario);
}
```

Base de datos

```
@Database(entities = {Usuario.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UsuarioDao usuarioDao();

    private static volatile AppDatabase INSTANCE;

    public static AppDatabase getInstance(Context context) {
        if (INSTANCE == null) {
            synchronized (AppDatabase.class) {
                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                            AppDatabase.class, "app_database")
                            .build();
                }
            }
        }
```

```
        return INSTANCE;
    }
}
```

Uso de Room

```
// Obtener la instancia de la base de datos
AppDatabase db = AppDatabase.getInstance(this);

// Insertar un usuario (debe hacerse en un hilo secundario)
new Thread(new Runnable() {
    @Override
    public void run() {
        Usuario usuario = new Usuario("Juan", "juan@example.com", 25);
        long id = db.usuarioDao().insert(usuario);
    }
}).start();

// Consultar todos los usuarios (debe hacerse en un hilo secundario)
new Thread(new Runnable() {
    @Override
    public void run() {
        List<Usuario> usuarios = db.usuarioDao().getAll();

        // Actualizar la UI en el hilo principal
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                // Hacer algo con la lista de usuarios
            }
        });
    }
}).start();
```

Conectividad y Threads

Hilos y procesos

En Android, todas las operaciones de la interfaz de usuario se ejecutan en el hilo principal (UI Thread). Las operaciones largas o bloqueantes deben ejecutarse en hilos secundarios para evitar que la interfaz se congele.

Thread

```
new Thread(new Runnable() {
    @Override
```

```
    public void run() {
        // Código a ejecutar en segundo plano

        // Actualizar la UI en el hilo principal
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                // Código para actualizar la UI
            }
        });
    }
}).start();
```

Handler

Los handlers permiten enviar y procesar mensajes y objetos Runnable asociados a la cola de mensajes de un hilo.

```
// Crear un handler asociado al hilo principal
Handler handler = new Handler(Looper.getMainLooper());

new Thread(new Runnable() {
    @Override
    public void run() {
        // Código a ejecutar en segundo plano

        // Enviar un runnable al hilo principal
        handler.post(new Runnable() {
            @Override
            public void run() {
                // Código para actualizar la UI
            }
        });
    }
}).start();
```

AsyncTask (Deprecated)

AsyncTask fue una clase para realizar operaciones en segundo plano y publicar resultados en el hilo de la UI. Aunque está obsoleta desde Android 11, aún se encuentra en muchos ejemplos y aplicaciones.

```
private class MiTarea extends AsyncTask<String, Integer, String> {
    @Override
    protected void onPreExecute() {
        // Código a ejecutar antes de iniciar la tarea en segundo plano
        // (se ejecuta en el hilo principal)
    }
```

```
    @Override
    protected String doInBackground(String... params) {
        // Código a ejecutar en segundo plano
        // params[0] contiene el primer parámetro

        // Publicar progreso
        publishProgress(50);

        return "Resultado";
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        // Actualizar el progreso en la UI
        // values[0] contiene el valor publicado
    }

    @Override
    protected void onPostExecute(String result) {
        // Código a ejecutar después de completar la tarea en segundo plano
        // (se ejecuta en el hilo principal)
        // result contiene el valor devuelto por doInBackground()
    }
}

// Ejecutar la tarea
new MiTarea().execute("Parámetro");
```

Alternativas modernas a AsyncTask

Executor

```
 Executor executor = Executors.newSingleThreadExecutor();
Handler handler = new Handler(Looper.getMainLooper());

executor.execute(new Runnable() {
    @Override
    public void run() {
        // Código a ejecutar en segundo plano

        // Actualizar la UI en el hilo principal
        handler.post(new Runnable() {
            @Override
            public void run() {
                // Código para actualizar la UI
            }
        });
    }
});
```

Coroutines (Kotlin)

```kotlin
// En una actividad o fragmento
private val coroutineScope = CoroutineScope(Dispatchers.Main)

fun realizarTareaEnSegundoPlano() {
    coroutineScope.launch {
        // Iniciar una coroutine en el hilo principal

        val resultado = withContext(Dispatchers.IO) {
            // Código a ejecutar en segundo plano
            "Resultado"
        }

        // De vuelta al hilo principal
        // Actualizar la UI con el resultado
    }
}
```

Servicios REST

Los servicios REST (Representational State Transfer) son una arquitectura de comunicación basada en HTTP que permite a las aplicaciones comunicarse con servidores web.

Características de REST

- **Arquitectura cliente-servidor**: Separación clara entre cliente y servidor.
- **Sin estado**: Cada petición contiene toda la información necesaria.
- **Cacheable**: Las respuestas pueden ser cacheadas.
- **Interfaz uniforme**: Recursos identificados por URLs, operaciones estándar (GET, POST, PUT, DELETE).
- **Sistema en capas**: El cliente no sabe si está conectado directamente al servidor final.

Verbos HTTP

- **GET**: Obtener recursos.
- **POST**: Crear recursos.
- **PUT**: Actualizar recursos.
- **DELETE**: Eliminar recursos.

Formatos de datos

Los formatos más comunes para intercambiar datos en servicios REST son:

- **JSON** (JavaScript Object Notation): Más ligero y fácil de parsear.
- **XML** (eXtensible Markup Language): Más estructurado pero más verboso.

Bibliotecas para consumir servicios REST

HttpURLConnection (Nativo)

```
new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            URL url = new URL("https://api.example.com/data");
            HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
            connection.setRequestMethod("GET");

            int responseCode = connection.getResponseCode();
            if (responseCode == HttpURLConnection.HTTP_OK) {
                BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
                StringBuilder response = new StringBuilder();
                String line;
                while ((line = reader.readLine()) != null) {
                    response.append(line);
                }
                reader.close();

                final String result = response.toString();

                // Actualizar la UI en el hilo principal
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        // Procesar el resultado
                    }
                });
            }
            connection.disconnect();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}).start();
```

OkHttp

```
 // Añadir la dependencia en build.gradle
// implementation 'com.squareup.okhttp3:okhttp:4.9.1'

OkHttpClient client = new OkHttpClient();

Request request = new Request.Builder()
        .url("https://api.example.com/data")
        .build();

client.newCall(request).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
        e.printStackTrace();
    }

    @Override
    public void onResponse(Call call, Response response) throws IOException {
        if (response.isSuccessful()) {
            final String result = response.body().string();

            // Actualizar la UI en el hilo principal
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    // Procesar el resultado
                }
            });
        }
    }
});
```

Retrofit

```
 // Añadir las dependencias en build.gradle
// implementation 'com.squareup.retrofit2:retrofit:2.9.0'
// implementation 'com.squareup.retrofit2:converter-gson:2.9.0'

// Definir la interfaz del servicio
public interface ApiService {
    @GET("data")
    Call<DataModel> getData();

    @POST("data")
    Call<ResponseModel> postData(@Body DataModel data);
}

// Crear la instancia de Retrofit
Retrofit retrofit = new Retrofit.Builder()
```

```
        .baseUrl("https://api.example.com/")
        .addConverterFactory(GsonConverterFactory.create())
        .build();

// Crear la implementación de la interfaz
ApiService apiService = retrofit.create(ApiService.class);

// Realizar una petición
Call<DataModel> call = apiService.getData();
call.enqueue(new Callback<DataModel>() {
    @Override
    public void onResponse(Call<DataModel> call, Response<DataModel> response)
{
        if (response.isSuccessful()) {
            DataModel data = response.body();
            // Procesar los datos
        }
    }

    @Override
    public void onFailure(Call<DataModel> call, Throwable t) {
        t.printStackTrace();
    }
});
```

Procesamiento de JSON

JSONObject/JSONArray (Nativo)

```
try {
    // Parsear un objeto JSON
    JSONObject jsonObject = new JSONObject(jsonString);
    String nombre = jsonObject.getString("nombre");
    int edad = jsonObject.getInt("edad");

    // Parsear un array JSON
    JSONArray jsonArray = jsonObject.getJSONArray("items");
    for (int i = 0; i < jsonArray.length(); i++) {
        JSONObject item = jsonArray.getJSONObject(i);
        String itemNombre = item.getString("nombre");
        // Procesar cada item
    }
} catch (JSONException e) {
    e.printStackTrace();
}
```

Gson

```
// Añadir la dependencia en build.gradle
// implementation 'com.google.code.gson:gson:2.8.8'

// Definir la clase modelo
public class Usuario {
    private String nombre;
    private int edad;
    private List<Item> items;

    // Getters y setters
}

public class Item {
    private String nombre;

    // Getters y setters
}

// Parsear JSON a objeto
Gson gson = new Gson();
Usuario usuario = gson.fromJson(jsonString, Usuario.class);

// Convertir objeto a JSON
String jsonString = gson.toJson(usuario);
```

Documentación, depuración e instalación

Documentación

La documentación del código es esencial para facilitar su mantenimiento y comprensión.

Javadoc

Javadoc es el estándar para documentar código Java.

```
/**
 * Esta clase representa un usuario en el sistema.
 *
 * @author Tu Nombre
 * @version 1.0
 */
public class Usuario {
    private String nombre;
    private int edad;
```

```
    /**
     * Constructor que crea un nuevo usuario.
     *
     * @param nombre El nombre del usuario
     * @param edad La edad del usuario
     */
    public Usuario(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    /**
     * Obtiene el nombre del usuario.
     *
     * @return El nombre del usuario
     */
    public String getNombre() {
        return nombre;
    }

    /**
     * Establece el nombre del usuario.
     *
     * @param nombre El nuevo nombre del usuario
     */
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    /**
     * Comprueba si el usuario es mayor de edad.
     *
     * @return true si el usuario es mayor de edad, false en caso contrario
     */
    public boolean esMayorDeEdad() {
        return edad >= 18;
    }
}
```

Depuración

Android Studio proporciona varias herramientas para depurar aplicaciones.

Depurador

El depurador permite ejecutar el código paso a paso, establecer puntos de interrupción y examinar variables.

Para establecer un punto de interrupción, haz clic en el margen izquierdo del editor de código junto a la línea donde quieres detener la ejecución.

Logcat

Logcat es una herramienta que muestra los mensajes de registro del sistema y de la aplicación.

```
 // Diferentes niveles de log
Log.v(TAG, "Mensaje de nivel VERBOSE");
Log.d(TAG, "Mensaje de nivel DEBUG");
Log.i(TAG, "Mensaje de nivel INFO");
Log.w(TAG, "Mensaje de nivel WARNING");
Log.e(TAG, "Mensaje de nivel ERROR");

// Con formato
Log.d(TAG, "Usuario: " + usuario.getNombre() + ", Edad: " + usuario.getEdad());
Log.d(TAG, String.format("Usuario: %s, Edad: %d", usuario.getNombre(),
usuario.getEdad()));
```

Layout Inspector

Layout Inspector permite examinar la jerarquía de vistas de la aplicación en tiempo de ejecución.

Profiler

Android Profiler proporciona información en tiempo real sobre el uso de CPU, memoria y red de la aplicación.

Device File Explorer

Device File Explorer permite explorar el sistema de archivos del dispositivo o emulador.

Instalación y distribución

Compilación de la aplicación

El proceso de compilación de una aplicación Android incluye:

1. Compilación del código Java/Kotlin a bytecode.
2. Conversión del bytecode a DEX (Dalvik Executable).
3. Empaquetado de DEX, recursos y archivos de configuración en un APK o AAB.
4. Firma del APK/AAB.
5. Alineación del APK para optimizar el uso de memoria.

Firma de la aplicación

La firma de la aplicación garantiza su autenticidad e integridad.

En Android Studio: 1. Build > Generate Signed Bundle/APK 2. Seleccionar APK o Android App Bundle 3. Crear o seleccionar un keystore 4. Completar la información del keystore y la clave 5. Seleccionar la variante de compilación (debug, release) 6. Finalizar

Distribución

Las principales formas de distribuir una aplicación Android son:

1. **Google Play Store**: La tienda oficial de aplicaciones de Android.
2. **Otras tiendas**: Amazon Appstore, Samsung Galaxy Store, etc.
3. **Distribución directa**: Compartir el APK directamente o a través de un sitio web.

Para subir una aplicación a Google Play: 1. Crear una cuenta de desarrollador en Google Play Console. 2. Crear una nueva aplicación. 3. Completar la información de la ficha de la aplicación. 4. Subir el APK o AAB firmado. 5. Configurar precios y distribución. 6. Publicar la aplicación.

Permisos

Los permisos en Android protegen la privacidad del usuario, restringiendo el acceso a datos y funciones sensibles.

Tipos de permisos

- **Permisos normales**: No representan un riesgo para la privacidad del usuario. Se conceden automáticamente.
- **Permisos peligrosos**: Pueden afectar a la privacidad del usuario. Requieren aprobación explícita.
- **Permisos de firma**: Solo se conceden a aplicaciones firmadas con el mismo certificado que la aplicación que declara el permiso.
- **Permisos de sistema**: Solo se conceden a aplicaciones preinstaladas en la imagen del sistema.

Declaración de permisos

Los permisos se declaran en el archivo AndroidManifest.xml:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.app">
```

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

    <application
        ...
    </application>
</manifest>
```

Solicitud de permisos en tiempo de ejecución

Desde Android 6.0 (API 23), los permisos peligrosos deben solicitarse en tiempo de ejecución:

```
 // Comprobar si el permiso está concedido
if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
        != PackageManager.PERMISSION_GRANTED) {

    // Comprobar si debemos mostrar una explicación
    if (ActivityCompat.shouldShowRequestPermissionRationale(this,
Manifest.permission.CAMERA)) {
        // Mostrar una explicación al usuario
        AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setTitle("Permiso de cámara")
                .setMessage("Necesitamos acceso a la cámara para tomar fotos.")
                .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        // Solicitar el permiso
                        ActivityCompat.requestPermissions(MainActivity.this,
                                new String[]{Manifest.permission.CAMERA},
                                REQUEST_CAMERA_PERMISSION);
                    }
                })
                .show();
    } else {
        // Solicitar el permiso directamente
        ActivityCompat.requestPermissions(this,
                new String[]{Manifest.permission.CAMERA},
                REQUEST_CAMERA_PERMISSION);
    }
} else {
    // El permiso ya está concedido, proceder
    abrirCamara();
}
```

```java
// Manejar la respuesta del usuario
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);

    if (requestCode == REQUEST_CAMERA_PERMISSION) {
        if (grantResults.length > 0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
            // Permiso concedido
            abrirCamara();
        } else {
            // Permiso denegado
            Toast.makeText(this, "No se puede acceder a la cámara sin permiso",
Toast.LENGTH_SHORT).show();
        }
    }
}
```

Mejores prácticas para los permisos

- Solicitar solo los permisos necesarios para el funcionamiento de la aplicación.
- Solicitar los permisos en el momento en que se necesitan, no todos al inicio.
- Explicar al usuario por qué se necesita cada permiso.
- Manejar correctamente la denegación de permisos, proporcionando alternativas cuando sea posible.
- Comprobar los permisos cada vez que se vaya a realizar una operación que los requiera.

Resumen de conceptos clave

1. Conceptos Fundamentales de Android

- **Arquitectura Android**: Sistema operativo basado en Linux con capas: Kernel, HAL, Bibliotecas nativas, Framework de aplicaciones y Aplicaciones.
- **Componentes principales**: Activities, Services, Broadcast Receivers, Content Providers.
- **Ciclo de vida de aplicación**: Controlado por el sistema según recursos disponibles y necesidades del usuario.
- **Android Studio**: IDE oficial para desarrollo Android, basado en IntelliJ IDEA.
- **Estructura de proyecto**: app/ (código y recursos), gradle/ (scripts de configuración), AndroidManifest.xml (configuración de la aplicación).

- **Gradle**: Sistema de construcción para automatizar compilación, pruebas y empaquetado.

2. Diseño de Interfaces

Layouts

- **LinearLayout**: Organiza elementos en una única fila o columna.
- **RelativeLayout**: Posiciona elementos en relación a otros o al contenedor.
- **ConstraintLayout**: Layout flexible para crear diseños complejos con jerarquía plana.
- **FrameLayout**: Diseñado para mostrar un único elemento o superponer elementos.
- **TableLayout**: Organiza elementos en filas y columnas.

Controles Básicos

- **TextView**: Muestra texto no editable.
- **EditText**: Permite introducir y editar texto.
- **Button/ImageButton**: Elementos interactivos para iniciar acciones.
- **ImageView**: Muestra imágenes.
- **CheckBox**: Permite seleccionar múltiples opciones.
- **RadioButton/RadioGroup**: Permite seleccionar una única opción de un grupo.
- **Spinner**: Desplegable para seleccionar un elemento de una lista.
- **ProgressBar/SeekBar**: Muestran progreso o permiten selecciones en un rango.

Controles Avanzados

- **ListView/RecyclerView**: Muestran listas de elementos con scroll.
- **Adaptadores**: Puente entre datos y vistas (ArrayAdapter, BaseAdapter, RecyclerView.Adapter).
- **Menús**: OptionsMenu (barra de acción), menú contextual, PopupMenu.
- **Diálogos**: AlertDialog, DatePickerDialog, TimePickerDialog.
- **Notificaciones**: Informan al usuario sobre eventos importantes.

3. Actividades e Intents

Actividades

- **Ciclo de vida**: onCreate(), onStart(), onResume(), onPause(), onStop(), onDestroy().
- **Estados**: Activa, Pausada, Detenida, Destruida.
- **Guardado de estado**: onSaveInstanceState() para preservar datos durante cambios de configuración.

Intents

- **Explícitos**: Especifican el componente exacto a iniciar.
- **Implícitos**: Solicitan una acción sin especificar el componente.
- **Paso de datos**: putExtra() para enviar, getXXXExtra() para recibir.
- **Objetos complejos**: Implementar Serializable o Parcelable.
- **Filtros de intents**: Definen qué intents puede manejar un componente.
- **PendingIntent**: Referencia a un intent que se ejecutará en el futuro.

4. Fragmentos

- **Ciclo de vida**: Similar al de actividades con métodos adicionales (onAttach, onCreateView, onViewCreated, onDetach).
- **Tipos**: Estáticos (definidos en XML) y dinámicos (añadidos mediante código).
- **Transacciones**: add(), replace(), remove(), hide(), show(), addToBackStack().
- **Comunicación**: Interfaces para comunicación entre fragmentos y actividades.
- **Adaptación a pantallas**: Permiten crear interfaces flexibles para diferentes tamaños de dispositivo.

5. Persistencia de Datos

Preferencias Compartidas

- **Almacenamiento**: Pares clave-valor para datos primitivos.
- **Operaciones**: getSharedPreferences(), edit(), putXXX(), apply()/commit().
- **Tipos soportados**: String, int, boolean, float, long, Set.

Almacenamiento de Archivos

- **Interno**: Privado para la aplicación, se elimina al desinstalar.
- **Externo**: Puede ser accesible por otras apps, permanece tras desinstalar.
- **Operaciones**: openFileOutput(), openFileInput(), getFilesDir(), getExternalFilesDir().
- **Recursos**: raw/ (compilados con ID) y assets/ (sin procesar).

SQLite

- **Estructura**: Contrato, DbHelper (onCreate, onUpgrade), operaciones CRUD.
- **Operaciones**: insert(), query(), update(), delete().
- **Cursor**: Interfaz para acceder a resultados de consultas.
- **Transacciones**: beginTransaction(), setTransactionSuccessful(), endTransaction().

Room

- **Componentes**: Entity (tablas), DAO (operaciones), Database (configuración).
- **Anotaciones**: @Entity, @PrimaryKey, @ColumnInfo, @Dao, @Query, @Insert, etc.
- **LiveData**: Permite observar cambios en los datos.
- **Relaciones**: @Relation, @Embedded, @ForeignKey para modelar relaciones entre entidades.

6. Conectividad y Procesos

Hilos y Procesos

- **UI Thread**: Hilo principal para operaciones de interfaz.
- **Hilos secundarios**: Para operaciones largas o bloqueantes.
- **Handlers**: Permiten comunicación entre hilos.
- **AsyncTask**: Clase para operaciones en segundo plano (deprecated).
- **Alternativas**: Executor, ThreadPoolExecutor, Coroutines (Kotlin).

Servicios REST

- **Arquitectura**: Recursos identificados por URLs, operaciones estándar, sin estado.
- **Verbos HTTP**: GET (leer), POST (crear), PUT (actualizar), DELETE (eliminar).
- **Bibliotecas**: HttpURLConnection, OkHttp, Retrofit, Volley.
- **JSON**: JSONObject/JSONArray (nativo), Gson, Jackson, Moshi.

7. Permisos

- **Tipos**: Normales (automáticos), peligrosos (requieren aprobación), de firma, de sistema.
- **Declaración**: en AndroidManifest.xml.
- **Solicitud en tiempo de ejecución**: checkSelfPermission(), requestPermissions() (desde Android 6.0).
- **Mejores prácticas**: Solicitar solo cuando sea necesario, explicar por qué, manejar denegaciones.

8. Documentación y Depuración

- **Herramientas**: Depurador, Logcat, Layout Inspector, Profiler, Device File Explorer.
- **Logging**: Log.v(), Log.d(), Log.i(), Log.w(), Log.e() para diferentes niveles.
- **Javadoc**: Documentación de código con comentarios especiales.
- **Pruebas unitarias**: JUnit para verificar componentes aislados.

9. Instalación y Distribución

- **Compilación**: Código → bytecode → DEX → APK/AAB → firma → alineación.
- **Firma**: Garantiza autenticidad e integridad del APK.
- **Google Play**: Plataforma oficial de distribución.
- **Instalación manual**: Mediante APK con FileProvider para Android 7.0+.

Ejemplos prácticos

Ver el archivo de ejemplos prácticos para implementaciones detalladas de:

1. **Aplicación de Gestión de Contactos**: Utilizando RecyclerView, adaptadores personalizados, actividades, fragmentos y persistencia de datos.
2. **Aplicación de Gestión de Países**: Mostrando información de países con imágenes, filtrado y detalles.

Estos ejemplos ilustran los conceptos clave de la asignatura y son similares a los ejercicios que aparecen en los exámenes.

Consejos para el examen

Preparación general

1. **Repasa todos los conceptos teóricos**: Asegúrate de entender los fundamentos de Android, el ciclo de vida de actividades y fragmentos, y los diferentes tipos de persistencia de datos.

2. **Practica con los ejemplos**: Implementa los ejemplos prácticos proporcionados en esta guía y modifícalos para asegurarte de que entiendes cómo funcionan.

3. **Revisa los exámenes anteriores**: Analiza los patrones y tipos de preguntas que suelen aparecer en los exámenes.

4. **Organiza tu tiempo**: El examen puede incluir varias partes, asegúrate de distribuir el tiempo adecuadamente.

Durante el examen

1. **Lee detenidamente las instrucciones**: Asegúrate de entender lo que se pide antes de empezar a programar.

2. **Planifica antes de codificar**: Haz un esquema mental o en papel de la estructura de la aplicación antes de empezar a escribir código.

3. **Prioriza la funcionalidad básica**: Asegúrate de que las funciones principales de la aplicación funcionan antes de añadir características adicionales.
4. **Comenta tu código**: Incluye comentarios que expliquen las partes más complejas de tu código.
5. **Verifica tu solución**: Si tienes tiempo, revisa tu código para detectar posibles errores o mejoras.

Temas frecuentes en los exámenes

Basándonos en el análisis de exámenes anteriores, estos son los temas que aparecen con mayor frecuencia:

1. **Diseño de interfaces**: Creación de layouts con diferentes tipos de controles.
2. **Actividades y fragmentos**: Implementación del ciclo de vida, comunicación entre componentes.
3. **Adaptadores personalizados**: Para mostrar datos en ListView o RecyclerView.
4. **Persistencia de datos**: Uso de SQLite, Room, preferencias compartidas o archivos.
5. **Intents**: Navegación entre actividades y paso de datos.
6. **Hilos y procesos**: Operaciones en segundo plano para no bloquear la interfaz de usuario.
7. **Servicios REST**: Consumo de APIs y procesamiento de JSON.
8. **Permisos**: Solicitud y manejo de permisos en tiempo de ejecución.

Errores comunes a evitar

1. **No manejar correctamente el ciclo de vida**: Olvidar guardar el estado de la aplicación en onSaveInstanceState() o no liberar recursos en onPause() u onDestroy().
2. **Realizar operaciones de red en el hilo principal**: Esto puede bloquear la interfaz de usuario y provocar un ANR (Application Not Responding).
3. **No validar la entrada del usuario**: Siempre verifica los datos introducidos por el usuario antes de procesarlos.
4. **Ignorar los permisos**: No solicitar los permisos necesarios o no manejar correctamente la denegación de permisos.

5. **Fugas de memoria**: No liberar recursos como Cursores, bases de datos o listeners cuando ya no son necesarios.

6. **No manejar excepciones**: Siempre incluye bloques try-catch para operaciones que puedan lanzar excepciones, especialmente en operaciones de red o de archivos.

7. **Hardcodear strings**: Utiliza recursos de strings (strings.xml) en lugar de hardcodear textos en el código.

Con esta guía y siguiendo estos consejos, estarás bien preparado para abordar el examen de junio de Programación Multimedia de Dispositivos Móviles. ¡Buena suerte!