T2-MOVILES.pdf

bloodyraintatii

Programación multimedia y dispositivos móviles

1º Desarrollo de Aplicaciones Multiplataforma

Estudios España

Accede al documento original

Programación multimedia

Tema 9: Servicios en Red II (teoría/práctica)

1. Conexiones HTTP y HTTPS
→ HTTP (Hypertext Transfer Protocol) → protocolo que permite conexiones remotas entre dispositivos, pero no cifra la información, lo que lo hace inseguro.
→ HTTPS (Hypertext Transfer Protocol Secure) → versión segura de HTTP que utiliza cifrado SSL/TLS para proteger la información transmitida, como contraseñas y otros datos sensibles.
→ HTTPS utiliza el puerto 443, y es el estándar actual para conexiones seguras en la web, especialmente en aplicaciones móviles y servicios en línea.

2. Esquema cliente/servidor en aplicaciones
→ Las aplicaciones Android que requieren conexión a un servidor externo siguen un esquema cliente/servidor.
→ Componentes del esquema:
- Aplicación Android → instalada en el dispositivo móvil.
- Servicio web basado en PHP → API REST que accede a la base de datos.
- Base de datos MySQL → almacena los datos en un servidor remoto.
- JSON → formato de intercambio de datos entre la aplicación y el servidor.

→ Funcionamiento:
- La aplicación realiza una petición HTTPS a la API REST a través de una URL.
- La API REST accede a la base de datos.
- La API REST devuelve los datos en formato JSON.
- La aplicación Android decodifica el JSON y procesa la información.

3. Permisos para conectar a Internet
→ Android utiliza un sistema de permisos para controlar el acceso de las aplicaciones a recursos como Internet, cámara, almacenamiento, etc.
→ Para que una aplicación pueda conectarse a Internet, es necesario declarar el permiso en el archivo *AndroidManifest.xml*.
→ Los permisos se solicitan al usuario durante la instalación de la aplicación.

4. Obtener JSON desde Android vía HTTP
→ Clases usadas → JSONParser (obtener datos en formato JSON de una URL específica) y ServidorPHPException (centralizamos la gestión de errores).
→ Métodos principales de la clase JSONParser:
- *getJSONArrayFromUrl* → obtiene un JSONArray desde una URL y un array con los parámetros POST.
- *getJSONObjectFromUrl* → obtiene un JSONObject desde “” “”.
- *buildURL* → construye una URL con parámetros para la petición.

→ Las conexiones HTTPS deben ejecutarse en un hilo separado para evitar bloquear la interfaz de usuario.
→ Se utiliza la biblioteca Volley para simplificar la obtención de JSON desde una URL.

5. Práctica: obtener JSON con la biblioteca Volley

→ Objetivo → utilizar la biblioteca Volley en una aplicación Android para realizar peticiones HTTP a un servidor remoto y obtener datos en formato JSON, procesándolos para su uso en la aplicación.

→ Pasos:

1. Añadir la dependencia de Volley al *build.gradle* del proyecto.
2. Crear una *RequestQueue* en la actividad principal para gestionar las peticiones de red eficientemente.
3. Realizar una petición HTTP utilizando *JsonObjectRequest* o *JsonArrayRequest*, dependiendo de si se espera un objeto o un array JSON como respuesta.
4. Configurar los métodos de respuesta para procesar los datos JSON recibidos, extrayendo la información necesaria utilizando métodos como *getString* o *getInt*.
5. Manejar errores de conexión o respuestas incorrectas del servidor en el método *onErrorResponse*.
6. Enviar parámetros en una petición POST creando un objeto *JSONObject* con los datos que se desean enviar al servidor y pasándolo en el cuerpo de la solicitud.
7. Optimizar el uso de la *RequestQueue* utilizando un patrón Singleton para evitar crear múltiples instancias y mejorar el rendimiento de la aplicación.
8. Considerar el uso de caché para almacenar respuestas y reducir el número de peticiones repetidas al servidor, mejorando la eficiencia de la aplicación.
9. Probar la aplicación realizando peticiones a una API real y verificando que los datos JSON se procesan correctamente.

Tema 10: Servicios en Red III (solo practica)

1. Práctica: Aprender cómo crear un proyecto de Google Maps

→ Objetivo → integrar Google Maps en una aplicación Android, configurando un proyecto en la Consola de Desarrolladores de Google y utilizando la API de Google Maps para mostrar mapas y realizar operaciones básicas como agregar marcadores.

→ Pasos:

1. Acceder a la Consola de Desarrolladores de Google desde una cuenta de Gmail y crear un nuevo proyecto.
2. Seleccionar la API de Google Maps (Maps SDK for Android) desde la biblioteca de APIs disponibles en el proyecto creado.
3. Generar una Clave de API en la sección de credenciales, la cual se utilizará para autenticar la aplicación y permitir el acceso a los servicios de Google Maps.
4. Instalar la SDK de Google Play Services en Android Studio, asegurándose de que esté disponible para todos los proyectos.
5. Añadir la dependencia de Google Maps en el archivo *build.gradle* del proyecto para incluir la funcionalidad de mapas.
6. Modificar el archivo *AndroidManifest.xml* para incluir la clave de API generada y otros elementos necesarios, como la versión de OpenGL ES.
7. Agregar un *MapFragment* en el archivo de diseño (XML) de la actividad donde se desee mostrar el mapa.
8. Implementar la interfaz *OnMapReadyCallback* en la actividad principal para manejar la carga del mapa y obtener una referencia al objeto *GoogleMap*.
9. Configurar el mapa en el método *onMapReady*, donde se pueden personalizar opciones como el tipo de mapa (satélite, híbrido, etc.) o agregar marcadores.
10. Ejecutar la aplicación en un dispositivo o emulador para verificar que el mapa se muestra correctamente y funciona según lo configurado.

Tema 11: Introducción a los videojuegos (teoría)

1. Los videojuegos. Concepto

→ Videojuego → mecanismo electrónico que interactúa con uno o varios jugadores para alcanzar objetivos o misiones, dando respuestas a las acciones del jugador.

→ Clasificación por plataformas → los videojuegos pueden ejecutarse en diversas plataformas como PC, PlayStation, Xbox, dispositivos móviles, consolas de Nintendo y máquinas recreativas.

→ Multiplataforma vs. exclusivos → algunos videojuegos están disponibles en varias plataformas (multiplataforma), mientras que otros son exclusivos de una sola plataforma.

→ Bloques funcionales → un videojuego se compone de varios bloques:

- Historia → hilo conductor del mundo representado.
- Arte conceptual → aspecto general del juego.
- Sonido → música, efectos, voces, sonidos ambientales.
- Mecánica de juego → funcionamiento general del juego.
- Diseño de programación → manera de implementación del juego en una máquina mediante un lenguaje y una metodología concretas.

→ Clasificación por género → los videojuegos se pueden clasificar en géneros como acción, aventura, lucha, plataformas, estrategia, puzzles y simuladores.

→ Actualizaciones y remakes → es común que los videojuegos reciban actualizaciones o remakes para mejorar su funcionamiento o adaptarlos a nuevas tecnologías.

2. Motores de videojuegos

→ Motor de videojuegos → parte central que soporta la lógica y funcionalidad del juego, permitiendo a los desarrolladores crear juegos sin necesidad de programar todo desde cero.

→ Componentes principales → un videojuego se divide en tres partes principales:

- El código → código para crear el juego en sí, con toda la lógica.
- El motor →
- Los recursos → sonido, imágenes, etc.

→ Los motores surgieron en los años 90 para soportar la complejidad de los videojuegos 3D, aunque también se utilizan en videojuegos 2D.

→ Abstracción del hardware → los motores permiten programar sin necesidad de conocer la arquitectura del hardware, gracias a APIs como OpenGL, DirectX o SDL.

→ Esta abstracción permite el desarrollo de aplicaciones multiplataforma.

→ Ventajas de usar motores:

- Facilitan la migración del videojuego a otras plataformas. Sin motor hay que cambiar el código de todas las plataformas si hay algún error y mirar compatibilidades.
- Simplifican la incorporación de nuevos programadores al proyecto. Sin motor hay que explicarles toda la estructura del código del proyecto.
- Permiten agregar nuevos efectos o funcionalidades más rápidamente. Sin motor hay que investigar y probar en todas las plataformas.
- Facilitan la creación de secuelas o nuevas versiones del videojuego. Sin motor hay que revisar todo el código para decidir qué podemos reutilizar y qué no.

3. Motores 2D

→ Definición de videojuegos 2D → son aquellos que utilizan solo dos dimensiones, alto y ancho, sin profundidad. Los elementos del juego se mueven en un plano.

→ Sprites → son representaciones gráficas de los elementos del juego (personajes, enemigos, etc.) que se superponen en capas.

→ El motor 2D permite dibujar figuras geométricas, escalar, rotar y recortar sprites, además de gestionar las capas del juego.

→ Los elementos del juego se organizan en capas, como el fondo, los personajes, los enemigos y la información del juego (vidas, tiempo, puntuación).

4. Motores 3D

→ Definición de videojuegos 3D → utilizan tres dimensiones, alto, ancho y profundidad, permitiendo movimientos libres en el espacio.

→ Renderizado → procesamiento de imágenes 3D a 2D para adaptarlas al ojo humano.

→ Modelado 3D → los objetos en 3D están formados por vértices que crean polígonos. A más vértices, mayor realismo, pero también mayor coste de procesamiento.

→ Se aplican texturas a los modelos 3D para darles realismo y se añaden luces y cámaras para crear escenas.

→ Se usan herramientas de modelado como Blender para crear modelos 3D de personajes y objetos.

→ Funciones del motor 3D → simplifica el proceso de creación, movimiento, rotación y escalado de modelos 3D, además de gestionar luces y cámaras.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

No son cuernos...

si tienes más de un banco
para tener aún más ventajas.

Abre también una **Cuenta NoCuenta** gratuita[1] y no te pierdas nada:

🔐 **Seguro de compra online gratis**[2] para tus pedidos.

✈️ Viaja y **ahorra en comisiones en el extranjero**[3].

🛍️ **Descuentos exclusivos** en tus marcas favoritas.

Saber más

[1]TIN 0 % y **TAE 0 %.**
[2]Compras superiores a 30 €
[3]Más info sobre el Plan de Viaje en ing.es

Tema 12: Desarrollo de videojuegos (teoría/práctica)

1. Desarrollo de un videojuego 2D. Interfaz y motor.
→ Interfaz y motor → para desarrollar un videojuego 2D, se comienza diseñando una interfaz que permita jugar y un motor básico que haga funcionar el juego.
→ Clase que herede de *SurfaceView* → se utiliza para crear una pantalla donde se dibujarán los elementos del juego, como si fuera un lienzo. Esta clase sobrescribe el método *onDraw*, que se encarga de dibujar los elementos en la pantalla.
→ Se crea una clase que hereda de *Thread* y controla aspectos como:

- FPS (Frames Por Segundo) → velocidad a la que se moverá el juego.
- Un objeto de la clase *PantallaVideojuego*.
- *Running* → variable que indica si el juego está corriendo o no.
- Lienzo → variable de tipo Canvas que dibuja los elementos en la pantalla.
- Variables para controlar las pulsaciones en la pantalla.

2. Agregando los sprites.
→ Sprites son las imágenes que representan los movimientos de los personajes. En este caso, se utilizan sprites simples con movimientos básicos (derecha, izquierda, arriba, abajo).
→ Clase *Sprite* → se crea una clase para gestionar los sprites, con métodos para generar posiciones aleatorias, cambiar direcciones, dibujar el personaje y cargar el sprite.
→ Enumerado *Direccion* → define las posibles direcciones en las que se puede mover el personaje.

3. Moviendo al personaje
→ Funcionalidad del motor → se modifica el método *run* del motor para que pinte al personaje en la pantalla. Se bloquea el *Canvas* para que solo el motor pueda dibujar en él.
→ Botones de control → se añaden botones para mover al personaje en diferentes direcciones (arriba, abajo, izquierda, derecha) y para pausar el juego.
→ Estela del personaje → inicialmente, el personaje deja una estela al moverse, pero esto se puede solucionar más adelante.

4. Agregado de fondo
→ Tipos de fondo → puede ser un color sólido o una imagen.
→ Se dibuja el fondo en el método *onDraw* de la clase *PantallaVideojuego*, antes de dibujar los personajes, para que no queden ocultos.
→ Se puede usar una imagen que se repita para cubrir toda la pantalla → se utilizan las clases *Bitmap* y *BitmapDrawable* para repetir la imagen de fondo y cubrir toda la pantalla.

5. Agregando los enemigos
→ Se agrega un enemigo con un sprite similar al del personaje, pero con movimientos aleatorios en el caso expuesto en el tema.
→ Implementación → se crea un objeto de tipo *Sprite* para el enemigo y se dibuja en el método *onDraw*. También se gestiona su movimiento aleatorio.
→ Mejoras → se sugiere agregar sonidos, como el sonido de aplastar a un enemigo, utilizando clases como *MediaPlayer* y *SoundPool*.

6. Concepto de animación en el desarrollo de videojuegos 3D

→ Los videojuegos en 3D son más complejos que los 2D por la dimensión de profundidad.

→ Conceptos básicos para la creación de un juego en 3D:

- Mundo → espacio donde se desarrolla el juego.
- Sistema de coordenadas → tres coordenadas (X, Y, Z) que representan alto, ancho y profundidad.
- Vectores → representan la posición exacta de un objeto en el mundo 3D.
- Componentes → en lugar de sprites, se utilizan vértices, polígonos y mallas de polígonos para crear modelos 3D.
- Grafo de escena → es una estructura que contiene todos los elementos del juego en forma de árbol, similar a cómo se dibujan los elementos en un motor 2D.

7. Práctica: creación de un videojuego

→ Objetivo → crear un videojuego.

→ Pasos:

1. Crear la interfaz y el motor:
    a. Diseñar una interfaz utilizando la clase *SurfaceView* para dibujar los elementos del juego.
    b. Crear una clase que herede de *Thread* para gestionar el motor del juego, controlando aspectos como los FPS y el estado del juego.
2. Agregar los sprites:
    a. Crear una clase *Sprite* para gestionar los movimientos del personaje.
    b. Utilizar un enumerado *Direccion* para definir las posibles direcciones de movimiento.
3. Mover al personaje:
    a. Implementar la funcionalidad en el motor para que el personaje se mueva en la pantalla.
    b. Añadir botones de control para mover al personaje en diferentes direcciones y pausar el juego.
4. Agregar el fondo:
    a. Dibujar un fondo en el método *onDraw* de la clase *PantallaVideojuego*.
    b. Utilizar una imagen que se repita para cubrir toda la pantalla.
5. Agregar enemigos:
    a. Crear un objeto *Sprite* para el enemigo y gestionar su movimiento aleatorio.
    b. Dibujar al enemigo en el método *onDraw* después del fondo.
6. Mejoras opcionales:
    a. Agregar sonidos utilizando clases como *MediaPlayer* y *SoundPool*.

Tema 13: Introducción al lenguaje de Swift (teoría)

1. Introducción a Swift

→ Swift → lenguaje oficial para desarrollar aplicaciones móviles en dispositivos iOS y macOS.

→ Fue desarrollado por Apple y presentado en 2014 durante la WWDC (Worldwide Developers Conference).

→ Características:

- Es un lenguaje multiparadigma, soportando programación orientada a objetos, funcional, imperativa y orientada a protocolos.
- Es software libre y multiplataforma, lo que permite su uso en GNU/Linux, Windows y macOS.

→ Uso principal → desarrollo de aplicaciones para iOS, aunque también se puede utilizar para aplicaciones de macOS.

→ Anteriormente, se usaba Objective-C para desarrollar aplicaciones iOS, pero Swift está reemplazándolo rápidamente.

2. Variables y operaciones

→ Tipos de datos básicos:

- Int → números enteros de 32 bits.
- Double → números reales de 64 bits con hasta 15 decimales de precisión.
- Bool → valores booleanos (true o false).
- String → cadenas de caracteres.

Declaración de variables:

- Se usa la palabra reservada *var* para variables.
- Se usa *let* para constantes.
- Swift infiere el tipo de dato automáticamente, pero también se puede especificar manualmente.

Operaciones:

- Números → suma, resta, multiplicación, división y cálculo del resto.
- Cadenas → concatenación y otras operaciones propias de la clase *String*.

→ Las variables en Swift deben tener un valor inicial; no se permiten variables sin inicializar.

3. Entrada y salida de datos

→ Entrada de datos:

- Se utiliza la función *readLine()* para leer datos desde el teclado. Esta función devuelve un *String*.
- Para convertir la entrada a otros tipos de datos (como *Int* o *Double*), se realiza un casting.
- La función *readLine()* devuelve un valor opcional, por lo que se debe manejar con el operador *!* para desempaquetarlo.

→ Salida de datos:

- Se utiliza la función *print()* para mostrar información por pantalla.
- Se puede usar la interpolación de cadenas con *\(variable)* para incluir valores de variables dentro de una cadena.

→ Punto y coma → no es necesario terminar las instrucciones con punto y coma, excepto cuando se anidan múltiples instrucciones en una misma línea.

4. Arrays

→ Definición → colección de elementos del mismo tipo.

→ Declaración:

- Se declaran usando corchetes *[]* y separando los elementos con comas.
- Ejemplo: *var array = [1, 2, 3, 4, 5]*.
- También se puede especificar el tipo de elementos (tanto en variables como en constantes) → *var palabras: [String] = ["Hola", "que", "tal"]*.

→ Métodos comunes:

- *count* → devuelve el número de elementos en el array.
- *append(valor)* → añade un valor al final del array.
- *insert(valor, at: posición)* → inserta un valor en una posición específica.
- *remove(at: posición)* → elimina un elemento en una posición específica.
- *sort()* → ordena el array en orden ascendente.
- *sort(by: >)* → ordena el array en orden descendente.
- *reversed()* → invierte el orden de los elementos.
- *shuffle()* → mezcla los elementos de forma aleatoria (disponible a partir de Swift 4.2).

→ Se puede imprimir un array completo usando *print(array)*.

Tema 14: Programación en Swift (teoría)

1. Estructuras condicionales

→ If-else → permite ejecutar un bloque de código si una condición es verdadera. Si no, se ejecuta el bloque *else*. Se pueden usar operadores lógicos como *&&* (AND) y || (OR) para evaluar múltiples condiciones.

```
if numero % 2 == 0 {
    print("El número es par")
} else {
    print("El número es impar")
}
```

→ Switch → evalúa una variable y ejecuta diferentes bloques de código según su valor. Es obligatorio incluir un caso *default* para manejar valores no contemplados.

```
switch numero {
    case 1:
        print("Es uno")
    case 2:
        print("Es dos")
    default:
        print("Otro número")
}
```

2. Estructuras repetitivas

→ For → itera sobre un rango de valores o elementos de un array.

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

```
for valor in array {
    print("El valor es \(valor)")
}
```

→ también se puede usar para recorrer arrays

→ Repeat-while → ejecuta un bloque de código mientras se cumpla una condición.

```
repeat {
    print("Introduce un número par")
    let numero = Int(readLine()!)!
} while numero % 2 != 0
```

3. Métodos

→ Definición → se declaran con la palabra reservada *func*, seguida del nombre, parámetros y tipo de retorno.

```
func sumar(a: Int, b: Int) -> Int {
    return a + b
}
```

→ Llamada → si el método devuelve un valor, se asigna a una variable. Si no, se llama directamente.

```
let resultado = sumar(a: 3, b: 5)
print("El resultado es \(resultado)")
```

→ Parámetros → los nombres de los parámetros son obligatorios al llamar a la función.

4. Clases I. Structs e inicializadores

→ Clases → se definen con la palabra reservada *class*.

```
class Alumno {
    let nombre: String
    let edad: Int

    init(nombre: String, edad: Int) {
        self.nombre = nombre
        self.edad = edad
    }
}
```

→ Structs → son similares a las clases, pero no requieren inicialización de variables.

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

→ Inicializadores → se usan para crear objetos de una clase. Se definen con *init*.

```
init(nombre: String, edad: Int) {
    self.nombre = nombre
    self.edad = edad
}
```

→ Inicializadores de conveniencia → permiten crear múltiples inicializadores que llaman al inicializador.

```
convenience init(nombre: String) {
    self.init(nombre: nombre, edad: 18)
}
```

5. Tuplas

→ Tuplas → son agrupaciones de valores que pueden ser de diferentes tipos.

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

→ Acceso a elementos → se accede a los elementos de una tupla mediante índices o nombres.

```
print(tupla.0)  // Imprime "Hola"
print(tupla.2)  // Imprime -9.6
```

→ Tuplas con nombres → se pueden asignar nombres a los elementos de una tupla para acceder a ellos de manera más clara.

```
var tupla = (data1: "Hola", data2: 2020)
print(tupla.data1)  // Imprime "Hola"
```

→ Tuplas en funciones → las funciones pueden devolver tuplas para retornar múltiples valores.

```
func obtenerDatos() -> (nombre: String, edad: Int) {
    return ("Juan", 25)
}
```

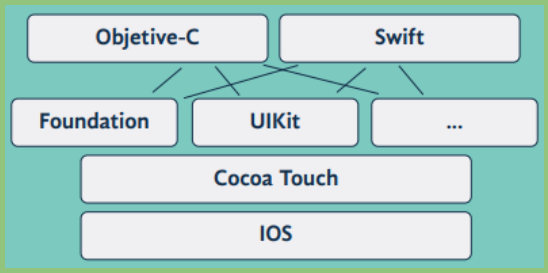
Tema 15: Introducción a MacOS y XCode (teoría)

1. Lenguajes de programación para aplicaciones en smartphones IOS

→ Lenguajes principales:

- Objective-C → antiguo lenguaje para desarrollar aplicaciones iOS, por lo que todavía se usa en proyectos antiguos. Está orientado a objetos y basado en C.
- Swift → es el lenguaje moderno y preferido para el desarrollo de aplicaciones iOS. Fue introducido por Apple en 2014 y ofrece varias ventajas sobre Objective-C. Es el lenguaje recomendado para nuevos proyectos.

→ API de Cocoa Touch → la interfaz de programación de aplicaciones (API) que permite desarrollar aplicaciones para dispositivos iOS. Incluye frameworks como *Foundation* y *UIKit*, que son esenciales para crear interfaces de usuario y gestionar la lógica de las aplicaciones.



→ Ventajas de Swift respecto a Objective-C:

- Código más conciso → menos líneas de código.
- Gestión automática de memoria → conteo automático de referencias (ARC) para gestionar la memoria, reduciendo errores comunes de gestión manual de memoria.
- Tipado fuerte de datos → ayuda a prevenir errores en tiempo de compilación y mejora la seguridad del código.

CRITERIOS

Tema 9: Servicios en Red II (teoría/práctica)
1. Conexiones HTTP y HTTPS
2. Esquema cliente/servidor en aplicaciones
3. Permisos para conectar a Internet
4. Obtener JSON desde Android vía HTTP
5. Práctica: obtener JSON con la biblioteca Volley

Tema 10: Servicios en Red III (solo practica)
1. Práctica: Aprender cómo crear un proyecto de Google Maps

Tema 11: Introducción a los videojuegos (teoría)
1. Los videojuegos. Concepto
2. Motores de videojuegos
3. Motores 2D
4. Motores 3D

Tema 12: Desarrollo de videojuegos (teoría/práctica)
1. Desarrollo de un videojuego 2D.
2. Agregando los sprites.
3. Moviendo al personaje
4. Agregado de fondo
5. Agregando los enemigos
6. Concepto de animación en el desarrollo de videojuegos 3D
7. Práctica: creación de un videojuego

Tema 13: Introducción al lenguaje de Swift (teoría)
1. Introducción a Swift
2. Variables y operaciones
3. Entrada y salida de datos
4. Arrays

Tema 14: Programación en Swift (teoría)
1. Estructuras condicionales
2. Estructuras repetitivas
3. Métodos
4. Clases I. Structs e inicializadores
5. Tuplas

Tema 15: Introducción a MacOS y XCode (teoría)
1. Lenguajes de programación para aplicaciones en smartphones IOS

Práctica
Obtener JSON con la biblioteca Volley
Aprender cómo crear un proyecto de Google Maps
Creación de un videojuego