Universidad Icesi

Seminario de Software

Semanas 9 a 16: Desarrollo Frontend Multiplataforma y Asistido por IA

Facultad de Ingeniería, Diseño y Ciencias Aplicadas — Universidad Icesi

Universidad ICESI

- Unidades 3 & 4
- De Java a Flutter
- Caso Contador
- 4 Enfoques
- Próximos Pasos

Ruta de Aprendizaje — Sesión 17

Contexto del Curso y Unidades 3 & 4

Propósito general, cronograma de sesiones (17 a 32) e hitos evaluativos con IA.

De Java y APO al Frontend Moderno

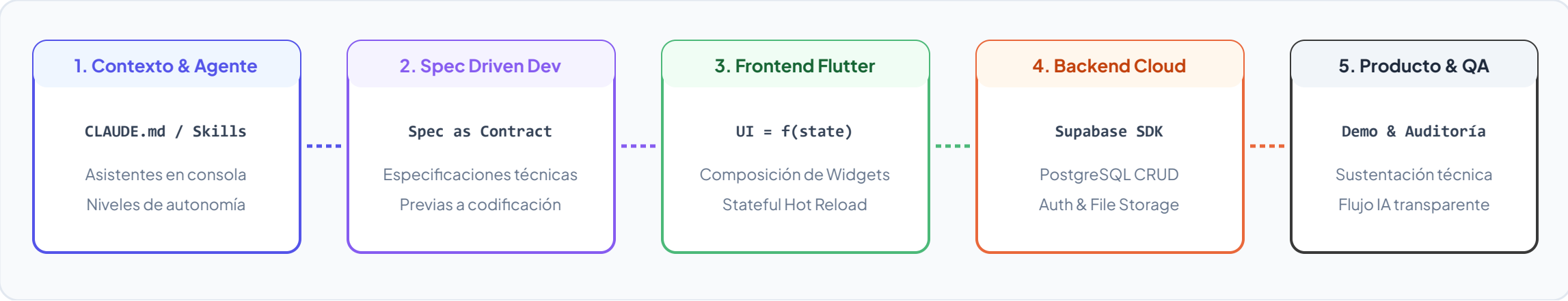
Los 3 cambios conceptuales: Declarativo, Composición y Compilación Nativa AOT.

Caso Práctico: Contador Interactivo

Mutación manual en JavaFX vs Reconstrucción por Estado con setState() en Flutter.

El Flujo Contemporáneo de Desarrollo con IA

Capacitar en el **flujo contemporáneo de desarrollo asistido por IA**: desde la configuración de contexto y agentes en consola, pasando por la construcción de interfaces declarativas en Flutter, hasta la integración con backend cloud (Supabase) bajo especificaciones rigurosas (SDD).



Articulación de las Unidades 3 y 4

UNIDAD 3: DESARROLLO ASISTIDO POR IA

Resultado de Aprendizaje (RA3): Configurar asistentes inteligentes, archivos de contexto, herramientas y protocolos de extensión para desarrollar software de forma productiva y controlada.

- Asistentes en consola y niveles de autonomía.
- Archivos de contexto (`CLAUDE.md`) como contratos vivos.
- Creación e instalación de *skills* especializadas.
- Especificaciones técnicas previas al código (*SDD*).
- Arquitectura cliente-servidor con servidores MCP.

UNIDAD 4: FRONTEND MULTIPLATAFORMA

Resultado de Aprendizaje (RA4): Desarrollar una aplicación frontend multiplataforma asistida por IA e integrada a servicios cloud, aplicando patrones declarativos y manejo de estado.

- Paradigma declarativo: la interfaz como función del estado.
- Composición con widgets Stateless y Stateful.
- Elevación de estado (*State Lifting*) y modelos inmutables.
- Navegación declarativa por rutas y pestañas (Tabs).
- Integración con Supabase (Auth, Postgres y Storage).

Universidad ICESI

Cronograma de Sesiones (Semanas 9 a 16)

SEM 9-10: SETUP & MAQUETACIÓN

- **Sesión 17–18**: Multiplataforma, SDK, emulador, widgets básicos y Stateless.
- **Sesión 19–20**: Scaffold, Row/Column, Expanded y agentes en consola (`CLAUDE.md`).

SEM 11-12: ESTADO & ROUTING

- **Sesión 21–22**: Skills de Flutter, maquetas, StatefulWidget y ciclo de vida.
- **Sesión 23–24**: Elevación de estado, callbacks, modelos y Navigator con rutas nombradas.

SEM 13-14: SDD & SUPABASE DB

- **Sesión 25**: **Entrega Parcial 1** (Prototipo navegable y Tabs).
- **Sesión 26–28**: Spec Driven Dev (SDD), integración Supabase SDK y CRUD Postgres.

SEM 15-16: STORAGE & SUSTENTACIÓN

- **Sesión 29–30**: Supabase Auth, subida de archivos en Storage y cierre técnico.
- **Sesión 31–32**: **Entrega Final (50%)**: Sustentaciones y auditoría agéntica.

Hitos de Evaluación y Niveles de IA

Hito de Evaluación	Sesión / Semana	Descripción del Entregable	Nivel de IAG
1. Diseño y Modelo	Semana 11 / Sesión 21	Propuesta formal de la aplicación, diseño no funcional en Stitch/Figma y modelo de datos relacional.	Nivel 5 Exploración
2. Entrega Parcial 1	Semana 13 / Sesión 25	Prototipo navegable en Flutter con componentes modulares, navegación por tabs y estado local simulado.	Nivel 5 Exploración
3. Specs y Backend	Semanas 14–15 / Sesiones 26 a 30	Especificaciones técnicas completas (*SDD*) e integración funcional con Supabase (Auth, Postgres y Storage).	Nivel 4 y 5 Guiado
4. Entrega Final (50%)	Semana 16 / Sesiones 31 y 32	Sustentación técnica: Demostración en vivo de la app, repositorio, `CLAUDE.md`, skills y auditoría de prompts.	Nivel 5 Total

* Los niveles de IAG reflejan la autonomía y el uso estratégico de herramientas asistidas bajo contratos verificables.

Universidad ICESI

De Java y APO al Frontend Moderno

Universidad ICESI

Java POO

Tipado Estricto

JavaFX / Swing

El Punto de Partida: Java y APO

En los cursos de **Algoritmos y Programación Orientada a Objetos (APO 1 y APO 2)**, construiste las bases conceptuales de la ingeniería de software trabajando principalmente con **Java**:

- **Tipado estricto y seguro:** Variables, estructuras de datos y verificación estática.
- **Modelado orientado a objetos:** Clases, encapsulamiento, herencia e interfaces.
- **Interfaces gráficas de escritorio:** Creación de ventanas y paneles con **JavaFX** o **Swing**.

La evolución hacia el frontend moderno: No desechamos los principios de POO ni el rigor del tipado; damos el salto hacia *paradigmas declarativos, composición modular y compilación nativa*.

Paradigma Imperativo vs. Declarativo

Enfoque Imperativo (JavaFX)

Creas objetos visuales en memoria y programas manualmente los pasos exactos para modificarlos ante cada evento:

```
// Mutación manual explícita
btn.setOnAction(e -> {
    contador++;
    etiqueta.setText("Total: " + contador);
});
```

- El programador busca la referencia del nodo en la escena.
- Mutación directa de propiedades del componente.
- **Alto riesgo:** Desincronización entre datos y pantalla.

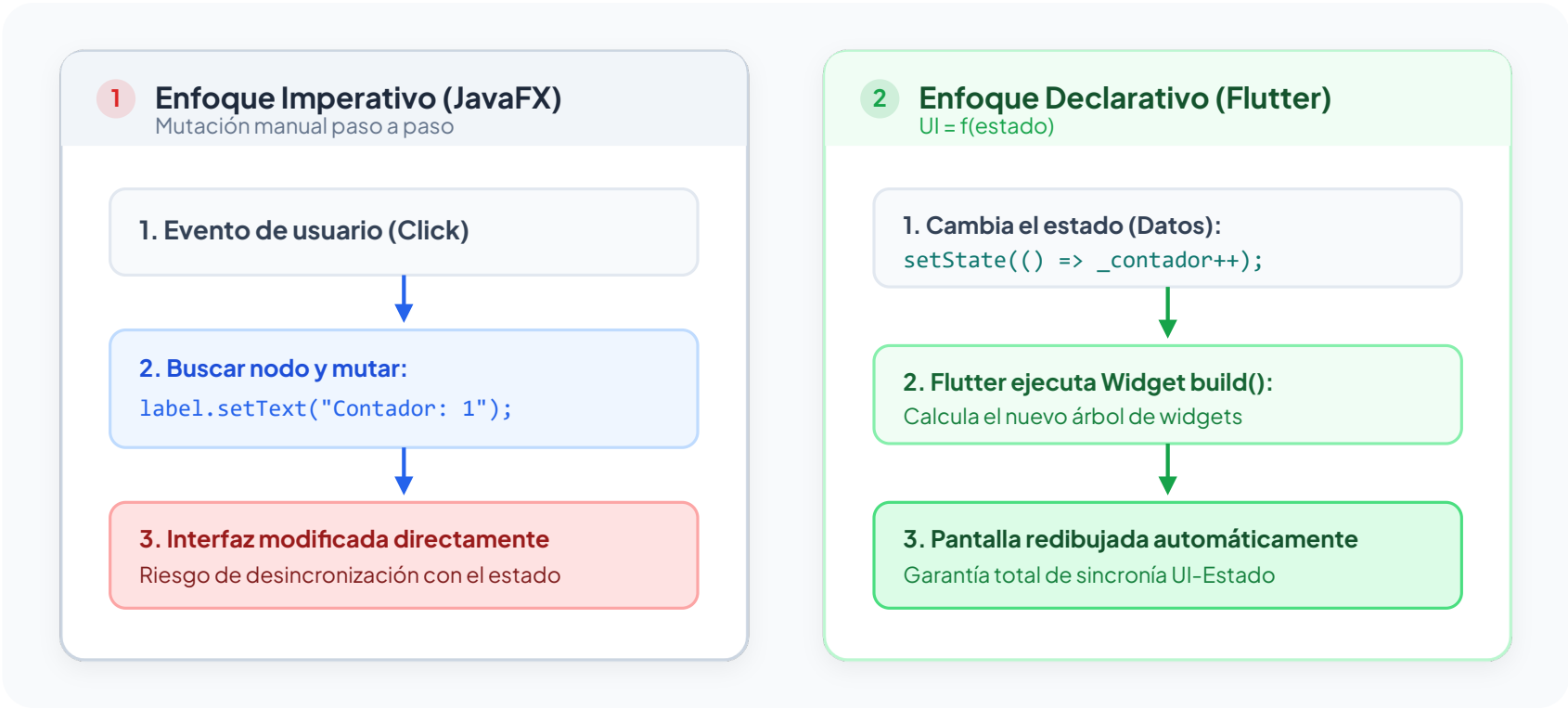
Enfoque Declarativo (Flutter)

La interfaz de usuario es el resultado matemático del estado actual:

$$UI = f(state)$$

- Describes cómo debe lucir la UI para un estado dado.
- Al cambiar los datos (`setState()`), Flutter re-evalúa y redibuja automáticamente los widgets necesarios.
- **Garantía:** Sincronía matemática total entre datos y UI.

Flujo de Ejecución: Imperativo vs Declarativo



Herencia en POO vs. Composición de Widgets

Herencia Clásica ("Es-Un")

En POO tradicional con Java, se acostumbra extender clases base para crear nuevos comportamientos visuales:

```
// Jerarquía vertical profunda
public class MiBotonConIcono extends Button {
    // Hereda cientos de métodos de Control y Node
}
```

- Genera cadenas rígidas de herencia (`Node > Control > ButtonBase`).
- Alto acoplamiento: cambios en la superclase rompen subclases.

Composición de Widgets ("Tiene-Un")

En Flutter se aplica el principio de **composición sobre herencia** combinando widgets pequeños:

```
// Cajas modulares anidadas
ElevatedButton(
  child: Row(
    children: [ Icon(Icons.save), Text("Guardar") ],
  ),
)
```

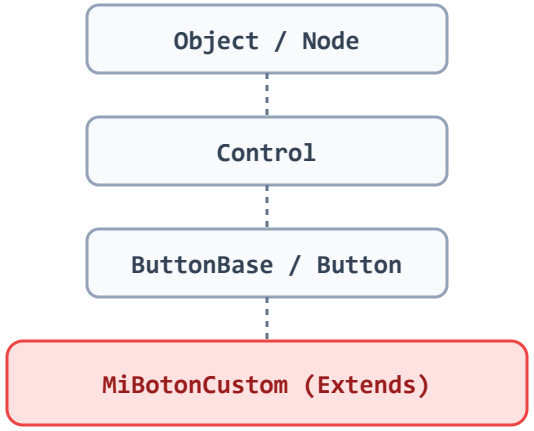
- Un botón no es una subclase especial; es un widget que contiene otros widgets.
- Máxima reusabilidad, modularidad y legibilidad.

Universidad ICESI

Estructura: Jerarquía Vertical vs Cajas Anidadas

Herencia Clásica en Java (POO)

Jerarquía Rígida



Alto acoplamiento y herencia profunda

Composición de Widgets en Flutter

Cajas Anidadas

```
ElevatedButton(
  child: Row(children: [
    Icono Visual
    Icon(Icons.save)
    Texto Etiqueta
    Text("Guardar")
  ])
```

Máxima modularidad, legibilidad y reusabilidad sin subclases

Modelo de Ejecución: JVM vs Flutter Dual

Java y la Máquina Virtual (JVM)

El código Java se compila a un formato intermedio independiente de la plataforma:

`.java` → **Bytecode (** `.class` **)** → **JVM en Runtime**.

- La JVM interpreta o compila JIT las instrucciones para el SO anfitrión.
- Capas intermedias pesadas entre la UI y el hardware.

Dart y Flutter: Modelo Dual

Combina lo mejor de dos mundos mediante dos modos de compilación especializados:

- **En Producción (Release) — Compilación AOT:**
  Compila anticipadamente a *código máquina nativo (ARM/x86)* directo a GPU a 60/120 fps.
- **En Desarrollo (Debug) — Stateful Hot Reload (JIT):**
  Inyecta cambios en < 1s *preservando el estado actual de la pantalla*.

Pipeline de Compilación: JVM vs Flutter Dual Engine

Modelo de Ejecución en Java (JVM)

Capa de Bytecode

`Código Fuente (.java)`

`Bytecode (.class / javac)`

Java Virtual Machine (JVM)
Intérprete + Compilador JIT en Runtime

Sistema Operativo / Hardware

Modelo Dual de Flutter (Dart)

AOT + JIT

Producción: Compilación AOT Nativa

`Dart → Código Máquina Nativo (ARM64 / x86)`

Dibujo directo a GPU (Impeller) a 60 / 120 fps

Desarrollo: Stateful Hot Reload (JIT)

Inyección incremental de código en < 1 segundo

Preserva el estado de la UI (formularios/tabs)
Iteración ultrarrápida sin reiniciar la aplicación

Universidad ICESI

Caso Práctico: Contador Interactivo

Contador en JavaFX: Mutación Imperativa

```java
public class ContadorApp extends Application {
    private int contador = 0; //1. Estado en memoria

    @Override
    public void start(Stage stage) {
        //2. Creación explícita de nodos UI
        Label lbl = new Label("Contador: 0");
        Button btn = new Button("Incrementar");

        //3. Mutación manual ante el evento
        btn.setOnAction(event -> {
            contador++;
            lbl.setText("Contador: " + contador);
        });

        VBox root = new VBox(10, lbl, btn);
        stage.setScene(new Scene(root, 300, 200));
        stage.setTitle("Contador JavaFX");
        stage.show();
    }
}
```

Puntos Críticos del Enfoque:

1. **Mutación Manual Obligatoria:**
   El desarrollador debe recordar cada nodo dependiente del dato y llamar explícitamente a `setText()`.
2. **Dispersión del Estado:**
   El valor vive duplicado: en la variable `contador` y dentro del nodo `lbl` en la escena gráfica.
3. **Riesgo de Inconsistencia:**
   Al escalar a múltiples vistas o paneles, olvidar actualizar una referencia produce pantallas desincronizadas.

Contador en Flutter: Estado y Reconstrucción

```dart
class ContadorWidget extends StatefulWidget {
  const ContadorWidget({super.key});
  @override
  State<contadorwidget> createState() => _ContadorWidgetState();
}

class _ContadorWidgetState extends State<contadorwidget> {
  int _contador = 0; //1. Estado local

  void _incrementar() {
    //2. setState notifica la reconstrucción
    setState(() => _contador++);
  }

  @override
  Widget build(BuildContext context) {
    //3. UI = f(estado)
    return Scaffold(
      appBar: AppBar(title: const Text('Contador Flutter')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text('Contador: $_contador', style: const TextStyle(fontSize: 22)),
            const SizedBox(height: 12),
            ElevatedButton(
              onPressed: _incrementar,
              child: const Text('Incrementar'),
            ),
          ],
        ),
      ),
    );
  }
}</contadorwidget></contadorwidget>
```

Ventajas del Enfoque Reactivo:

1. **Fuente Única de Verdad:**
   La variable `_contador` gobierna la pantalla. No existen referencias directas a componentes visuales.
2. **`setState()` Notificador:**
   Solo altera el dato y notifica al framework; Flutter re-ejecuta `build()` y redibuja eficientemente.
3. **Desacoplamiento Total:**
   Si añadimos gráficos, barras de progreso o etiquetas dependientes de `_contador`, se sincronizan solas.

Universidad ICESI

Beneficios del Enfoque Declarativo

PREVISIBILIDAD

Al ser `UI = f(state)`, un estado idéntico siempre produce exactamente la misma pantalla, eliminando inconsistencias visuales.

DESACOPLAMIENTO

La lógica de datos solo modifica variables de estado y desconoce por completo los detalles de renderizado en pantalla.

ESCALABILIDAD

Agregar nuevas vistas, animaciones o widgets que reaccionen al mismo dato no requiere reescribir manejadores de eventos.

TESTABILIDAD

Es trivial realizar pruebas unitarias sobre los cambios de estado sin tener que instanciar ventanas ni simular clics en el DOM/Escena.

Universidad ICESI

Los 4 Enfoques Multiplataforma

Universidad ICESI

Evolución de Enfoques de Ingeniería Frontend

1

Nativo Separado

Kotlin (Android) + Swift (iOS)

2 bases de código
Máximo acceso a APIs
Costo y tiempo duplicado

2

Híbrido WebView

Ionic / Cordova / Capacitor

Web embebida en app
HTML, CSS y JS estándar
Latencia táctil en animaciones

3

Puente Reactivo

React Native (Bridge / JSI)

Componentes nativos del SO
Lógica en JavaScript/TS
Sobrecarga en puente asíncrono

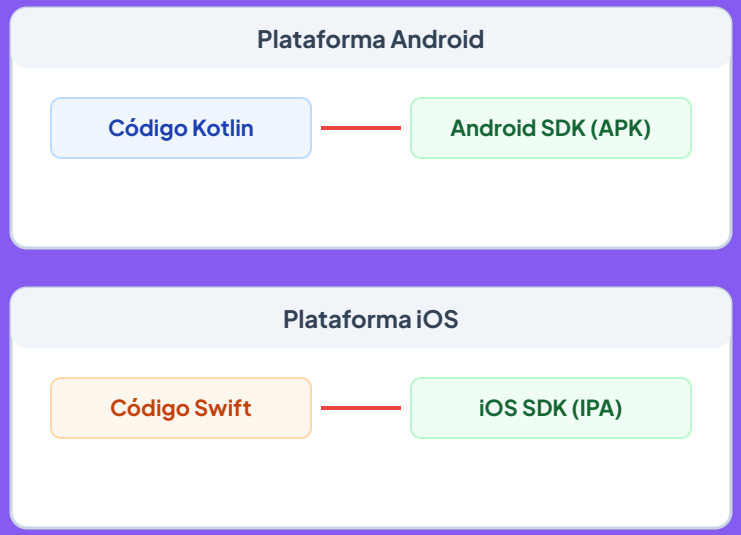
4

Motor Gráfico Propio

Flutter (Impeller / Skia)

Píxeles directos a GPU
Paridad visual 100% idéntica
Rendimiento fluido a 120 fps

Universidad ICESI



Nativo Tradicional (Código Separado)

Se construyen dos aplicaciones completamente independientes utilizando los lenguajes y herramientas oficiales de cada fabricante:

- **Android:** Kotlin / Java con Android Studio y Jetpack Compose / XML.
- **iOS:** Swift / Objective-C con Xcode y SwiftUI / UIKit.

Ventajas: Máximo rendimiento de cálculo, acceso inmediato el día 1 a nuevas APIs y SDKs del sistema.

Desventajas: Costo de desarrollo duplicado, necesidad de dos equipos especializados y alta probabilidad de discrepancias funcionales entre plataformas.

Híbrido Basado en WebViews

La aplicación es esencialmente un sitio web empaquetado dentro de un navegador embebido (**WebView**) que se ejecuta dentro del cascarón nativo:

- **Tecnologías:** Apache Cordova, Ionic, Capacitor.
- **Mecanismo:** Lógica e interfaz en HTML/CSS/JS con plugins para acceder a hardware.

Ventajas: 100% de reutilización del stack web tradicional (React, Angular o Vue) y despliegue rápido.

Desventajas: Rendimiento limitado en animaciones complejas, latencia al scroll táctil y apariencia no auténticamente nativa.

Puente Interpretado / Reactivo

La lógica se escribe en JavaScript/TypeScript, pero en lugar de una WebView, delega el dibujo en los **componentes nativos reales del SO** mediante un puente:

- **Tecnologías:** React Native (Meta).
- **Mecanismo:** Mapea `<View>` a `UIView` en iOS y `android.view.View` en Android.

Ventajas: Aspecto 100% nativo y acceso al gigantesco ecosistema de librerías de React.

Desventajas: Cuello de botella en el puente asíncrono durante animaciones rápidas y discrepancias de estilo entre iOS y Android.

Motor Propio (Lienzo GPU Directo)

Flutter no usa WebViews ni delega el dibujo en los widgets del SO. Incluye su propio motor gráfico 2D en C++ (**Impeller / Skia**) y dibuja cada píxel directamente en la GPU (similar a un motor de videojuegos):

- **Tecnologías:** Flutter & Dart (Google).
- **Mecanismo:** Dibuja en pantalla usando Metal en iOS y Vulkan en Android.

Ventajas: Paridad visual 100% idéntica en cualquier pantalla, rendimiento determinista a 120 fps y control de cada píxel.

Trade-off: Binario inicial ligeramente mayor al incluir el motor gráfico embebido.

Universidad ICESI

Flutter vs React Native en la Industria

Filosofía de los Líderes del Mercado

Flutter (Google)

"Control absoluto de cada píxel en pantalla"

- **Lenguaje:** Dart (tipado estricto, orientado a objetos, rápido aprendizaje desde Java).
- **Arquitectura:** Motor gráfico propio Impeller directo a GPU.
- **Consistencia:** Idéntica en iOS, Android, Web, Windows, macOS y Linux.
- **Productividad:** Stateful Hot Reload en < 1s.

React Native (Meta)

"Learn once, write anywhere"

- **Lenguaje:** JavaScript / TypeScript (ecosistema web masivo).
- **Arquitectura:** Mapeo a componentes nativos del SO vía JSI / Fabric.
- **Consistencia:** Cada plataforma adapta los estilos nativos propios de su sistema operativo.
- **Productividad:** Fast Refresh.

Matriz Comparativa Técnica

Criterio Técnico	Flutter (Google)	React Native (Meta)
Creador y Respaldo	Google	Meta (Facebook)
Lenguaje Principal	**Dart** (Tipado estricto, POO similar a Java)	**JavaScript / TypeScript** (Dinámico/Tipado)
Arquitectura de Render	Motor gráfico propio (**Impeller / Skia** directo a GPU)	Mapeo a widgets nativos del SO (**JSI / Fabric**)
Rendimiento de Animaciones	**Excelente** (Compilación AOT nativa a 60/120 fps)	**Muy bueno** (Hilo JS desacoplado del hilo UI)
Consistencia Visual	**100% idéntica** en cualquier pantalla	Varía según el estilo visual propio de cada SO
Experiencia de Desarrollo	**Stateful Hot Reload** (Conserva estado de UI)	**Fast Refresh**
Soporte Multiplataforma	Móvil (iOS/Android), Web y Desktop (Win/Mac/Linux)	Móvil (iOS/Android) — Web y Desktop vía comunidad

Próximos Pasos: Configuración de Flutter

FLUTTER SDK

- Descarga e instalación del SDK oficial de Flutter (versión estable).
- Configuración de la variable de entorno `PATH` del sistema.

HERRAMIENTAS & IDE

- Instalación de **VS Code** o **Android Studio**.
- Instalación de extensiones oficiales de **Dart** y **Flutter**.

EMULADOR & DISPOSITIVO

- Configuración de **Android Virtual Device (AVD)** o **iOS Simulator**.
- Habilitación de depuración USB en teléfono físico para pruebas.

FLUTTER DOCTOR & HOLA MUNDO

- Ejecución de `flutter doctor -v` para validar dependencias.
- Creación y ejecución de la primera app con `flutter create hola_mundo`.