Desarrollo de Interfaces

Unidad 07. Introducción a Dart

Autor: Sergi García

Actualizado Agosto 2025

Licencia

Nomenclatura

A lo largo de este tema se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

📖 **Importante**

❕ **Atención**

💬 **Interesante**

ÍNDICE

Unidad 07. Introducción a Dart

1. Introducción a Dart

1.1. ¿Por qué Dart?

🔷 Rendimiento: JIT y AOT

- **JIT (Just-In-Time)**
  Compila el código durante la ejecución, ideal para desarrollo:
  - Permite Hot Reload en Flutter: cambios al instante.
  - Iteraciones rápidas sin recompilar toda la app.
- **AOT (Ahead-Of-Time)**
  Compila a código máquina nativo antes de ejecutarse:
  - Arranque más rápido en producción.
  - Menor consumo de CPU y mejor rendimiento.

💬 Ejemplo visual del flujo:

Desarrollo: Dart → JIT → Ejecución rápida

Producción: Dart → AOT → Binario nativo → Ejecución optimizada

🔷 **Ecosistema Flutter**

- Dart es el lenguaje oficial de Flutter, framework de Google para crear apps móviles, web y escritorio con un solo código.
- Ventajas:
  - UI consistente en todas las plataformas.
  - Comunidad activa y creciente.
  - Integración nativa con herramientas como Dart DevTools.

🔷 **Simplicidad y Productividad**

- Sintaxis clara y moderna.
- Soporta null safety para evitar errores comunes.
- Herramientas integradas:
  - dart pub → gestión de paquetes.
  - dart compile → generar ejecutables.
- Combina programación orientada a objetos e influencias funcionales.

📖 Importante: Dart nació en 2011, pero su popularidad explotó en 2017 con la llegada de Flutter.

1.2. Dart vs Otros Lenguajes

💬 **Similitudes con JavaScript / TypeScript**

- Sintaxis moderna con var, final, const.
- Uso de async/await para asincronía.
- Funciones flecha (=>) y closures.
- Estructuras de datos similares: List, Map, Set.

📌 Ejemplo:

```
var nombres = ['Ana', 'Luis'];
```

```
nombres.forEach((n) => print(n));
```

💬 **Similitudes con Java**

- Tipado fuerte y estático (aunque puede inferirse).
- Clases, herencia, interfaces implícitas.
- Constructores con sobrecarga y this.
- Paquetes y estructura modular.

📌 Ejemplo:

```
class Persona {
  String nombre;
  Persona(this.nombre);
  void saludar() => print("Hola, soy $nombre");
}
```

❕ Atención:
Aunque Dart comparte elementos con JS y Java, se diferencia en:

- Null Safety integrada de serie.
- Compilación nativa sin puentes intermedios (mejor rendimiento).
- Mismo código para múltiples plataformas.

2. CONFIGURACIÓN Y HERRAMIENTAS

2.1. Instalación del SDK

🔧 Pasos generales:

1. Descarga desde la web oficial: https://dart.dev/get-dart
2. Extrae y añade la carpeta bin a la variable de entorno PATH.
3. Verifica instalación:

```
dart --version
```

📖 Importante:
El SDK incluye:

- dart pub → gestor de dependencias.
- dart compile → compilación a ejecutables nativos.

Ejemplo de compilación:

```
dart compile exe main.dart
```

2.2. IDEs y Plugins

💬 Opciones recomendadas:

- VS Code + extensión Dart.
- Android Studio + plugin Dart.
- DartPad (https://dartpad.dev) → editor online, sin instalar nada.

📖 Tip:
Usa VS Code para proyectos pequeños y Android Studio si vas a trabajar con Flutter.

2.3. Primer Proyecto

🚀 Crear un proyecto CLI:

```
dart create mi_proyecto
cd mi_proyecto
dart run
```

📦 Estructura de carpetas:

- bin/ → código principal.
- lib/ → librerías y módulos.
- pubspec.yaml → dependencias y metadatos.

💬 Atajo:
Puedes modificar bin/mi_proyecto.dart y ejecutar directamente:

```
dart run
```

3. Sintaxis Básica

3.1. Variables y Tipos

🔷 Declaración de variables

- var → Tipo inferido automáticamente.
- dynamic → Puede cambiar de tipo (⚠ evitar salvo casos concretos).
- final → Asignación única en tiempo de ejecución.
- const → Constante en tiempo de compilación.

📌 Ejemplo:

```dart
var nombre = "Ana";          // Inferido como String
dynamic dato = 42;           // Puede cambiar de tipo
final fecha = DateTime.now(); // Valor fijo tras asignarse
const pi = 3.1416;           // Constante de compilación
```

❕ Atención:
final y const no son lo mismo:

- final se fija en ejecución.

- const se fija en compilación.

🔷 Tipos básicos en Dart

- int → números enteros.
- double → números decimales.
- bool → verdadero/falso.
- String → texto.
- List → lista ordenada.
- Set → conjunto sin duplicados.
- Map → pares clave-valor.

💬 Ejemplo mixto:

```
int edad = 30;
double precio = 9.99;
bool activo = true;
String saludo = "Hola";
List<String> frutas = ["Manzana", "Pera"];
```

3.2. Operadores

🔷 **Operadores Null-aware**

- ?? → valor por defecto si es null.
- ?. → acceso seguro a propiedades (si no es null).
- !.. → ignora null safety (⚠ usar con cuidado).

📌 Ejemplo:

```
String? nombre;
print(nombre ?? "Desconocido"); // "Desconocido"

String? texto = "Hola";
print(texto?.length); // 4
```

🔷 **Cascade Operator (..)**

Permite encadenar operaciones sobre un mismo objeto sin repetir su nombre.

💬 Ejemplo:

```
var buffer = StringBuffer()
  ..write("Hola ")
  ..write("Mundo");
print(buffer.toString()); // "Hola Mundo"
```

3.3. Control de Flujo

🔷 **Condicionales**

```dart
if (edad >= 18) {
  print("Mayor de edad");
} else {
  print("Menor de edad");
}
```

🔷 Switch con Pattern Matching

Dart soporta patrones y casos múltiples:

```dart
var dia = "Lunes";

switch (dia) {
  case "Lunes":
  case "Martes":
    print("Inicio de semana");
    break;
  case "Sábado" || "Domingo":
    print("Fin de semana");
    break;
  default:
    print("Día normal");
}
```

📖 Importante:
Desde Dart 3, el switch permite pattern matching avanzado con tipos y condiciones.

🔷 Bucles

- for clásico:

```dart
for (var i = 0; i < 5; i++) {
  print(i);
}
```

- for-in (recorrer colecciones):

```dart
for (var fruta in ["Manzana", "Pera"]) {
  print(fruta);
}
```

- while / do-while:

```dart
var contador = 0;
while (contador < 3) {
  print(contador++);
```

```
}
```

💡 Consejo rápido:
Prefiere for-in y métodos como .forEach() para código más limpio al recorrer listas y conjuntos.

4. FUNCIONES AVANZADAS

4.1. Parámetros

🔷 Parámetros posicionales opcionales [ ]

- Se colocan entre corchetes.
- Se asigna un valor por defecto si no se recibe.

📌 Ejemplo:

```
void mostrarMensaje(String texto, [int veces = 1]) {
  for (var i = 0; i < veces; i++) {
    print(texto);
  }
}

mostrarMensaje("Hola");     // Muestra una vez
mostrarMensaje("Hola", 3);  // Muestra tres veces
```

🔷 **Parámetros nombrados { }**

- Se especifican por nombre al llamar la función.
- Son más legibles y permiten valores por defecto.
- Se puede usar required para obligar su paso.

📌 Ejemplo:

```
void crearUsuario({required String nombre, int edad = 18}) {
  print("Usuario: $nombre, Edad: $edad");
}

crearUsuario(nombre: "Ana");
crearUsuario(nombre: "Luis", edad: 25);
```

🔷 **Combinando ambos tipos**

```
void registrar(String id, {String? nombre, int edad = 0}) {
  print("ID: $id, Nombre: $nombre, Edad: $edad");
}

registrar("123", nombre: "Pedro");
```

💡 Tip:

Usa parámetros nombrados para funciones con muchos argumentos → mejora la claridad del código.

4.2. Funciones Anónimas y Closures

🔷 **Funciones Anónimas (Lambdas)**

- No tienen nombre.
- Se asignan a variables o se pasan como argumento.

📌 Ejemplo:

```
var sumar = (int a, int b) {
  return a + b;
};

print(sumar(3, 4)); // 7
```

🔷 **Closures**

- Una función que captura variables de su entorno.
- Permite que una función interna "recuerde" el estado externo.

📌 Ejemplo:

```
Function contador() {
  int cuenta = 0;
  return () {
    cuenta++;
    print(cuenta);
  };
}

var c = contador();
c(); // 1
c(); // 2
```

💬 En este ejemplo, la variable cuenta sigue existiendo incluso después de que la función contador() haya terminado.

4.3. Lexical Scope y Arrow Functions

🔷 **Lexical Scope**

- Las funciones en Dart usan ámbito léxico:
  - Una función interna puede acceder a variables de su función externa.

📌 Ejemplo:

```
void externa() {
  var mensaje = "Hola";
  void interna() {
```

```
    print(mensaje);
  }
  interna();
}

externa(); // Hola
```

🔷 **Arrow Functions (=>)**

- Forma reducida para funciones que retornan una sola expresión.

📌 Ejemplo:

```
int cuadrado(int x) => x * x;
print(cuadrado(5)); // 25
```

💡 Regla práctica:
Usa => para funciones cortas y expresivas; para lógica más compleja, mejor las llaves { }.

5. COLECCIONES Y NULL SAFETY

5.1. Listas, Sets y Maps

🔷 **List (listas ordenadas)**

- Permiten elementos repetidos.
- Indexadas desde 0.

📌 Ejemplo:

```
var frutas = ["Manzana", "Pera", "Plátano"];
frutas.add("Uva");           // Agregar elemento
print(frutas[0]);            // Acceder por índice
print(frutas.length);        // Tamaño de la lista
```

💬 **Métodos útiles:**

```
// map → transforma elementos
var enMayus = frutas.map((f) => f.toUpperCase());

// where → filtra elementos
var filtradas = frutas.where((f) => f.startsWith("P"));

// fold → reduce a un valor
var totalLetras = frutas.fold(0, (sum, f) => sum + f.length);
```

🔷 **Set (conjuntos no repetidos)**

- No permite duplicados.
- Ideal para valores únicos.

📌 **Ejemplo:**

```
var numeros = {1, 2, 3, 3};
print(numeros); // {1, 2, 3} → el duplicado se ignora
numeros.add(4);
```

🔷 **Map (pares clave-valor)**

- Similar a un diccionario.
- Claves únicas.

📌 Ejemplo:

```
var persona = {
  "nombre": "Ana",
  "edad": 30
};

print(persona["nombre"]); // Ana
persona["edad"] = 31;     // Modificar valor
```

💡 Tip:
Usa Map<String, dynamic> si quieres mezclar tipos en los valores.

5.2. Null Safety

📖 Importante:
Dart usa null safety para evitar errores en tiempo de ejecución al acceder a variables que pueden ser null.

🔷 **Tipos anulables (?)**

- Un tipo anulable se indica con ? al final.

```
String? nombre = null;  // Puede ser null
```

🔷 **Operador !**

- Indica al compilador que estás seguro de que la variable no es null.
  ⚠ Si te equivocas, lanza error en tiempo de ejecución.

```
String? texto = "Hola";
print(texto!.length); // OK
```

🔷 **Operador ?.**

- Accede a propiedades solo si no es null.
- Si es null, retorna null sin lanzar error.

```
String? saludo;
print(saludo?.length); // null
```

🔷 **Operador ??**

- Devuelve un valor por defecto si es null.

```
String? usuario;
print(usuario ?? "Invitado"); // Invitado
```

🔷 **late variables**

- Se inicializan más tarde pero se garantiza que tendrán valor antes de usarse.

```
late String nombre;
nombre = "Luis";
print(nombre);
```

💬 **Ejemplo combinado:**

```
String? nombre;
print((nombre ?? "Desconocido").toUpperCase());
```

6. POO EN DART

6.1. Clases

🔷 **Definición básica**

```
class Persona {
  String nombre;
  int edad;

  // Constructor
  Persona(this.nombre, this.edad);

  // Método
  void saludar() {
    print("Hola, soy $nombre y tengo $edad años");
  }
}

void main() {
  var p = Persona("Ana", 25);
  p.saludar(); // Hola, soy Ana y tengo 25 años
}
```

🔷 Constructores con nombre

- Permiten crear constructores alternativos.

```
class Punto {
  double x, y;

  Punto(this.x, this.y);
  Punto.origen() : x = 0, y = 0;
}

var p1 = Punto(5, 3);
var p2 = Punto.origen();
```

🔷 Constructores factory

- Devuelven una instancia existente o personalizada.

```
class Conexion {
  static final Conexion _instancia = Conexion._interna();
  factory Conexion() => _instancia;
  Conexion._interna();
}

var c1 = Conexion();
var c2 = Conexion();
print(identical(c1, c2)); // true
```

🔷 Getters y Setters

```
class Rectangulo {
  double ancho, alto;

  Rectangulo(this.ancho, this.alto);

  double get area => ancho * alto; // getter
  set cambiarAncho(double valor) => ancho = valor; // setter
}

var r = Rectangulo(3, 4);
print(r.area); // 12
r.cambiarAncho = 5;
```

6.2. Herencia y Mixins

🔷 Herencia (extends)

```
class Animal {
  void hacerSonido() => print("Sonido genérico");
}

class Perro extends Animal {
  @override
  void hacerSonido() => print("Guau");
}

var perro = Perro();
perro.hacerSonido(); // Guau
```

🔷 Mixins (with)

- Añaden funcionalidades sin heredar.

```
mixin Volador {
  void volar() => print("Estoy volando");
}

class Pajaro with Volador {}

var p = Pajaro();
p.volar();
```

🔷 Restricción de mixins (on)

```
mixin Nadador on Animal {
  void nadar() => print("Estoy nadando");
}
```

6.3. Interfaces Implícitas y Clases Abstractas

🔷 Interfaces implícitas

- Todas las clases definen automáticamente una interfaz.
- Se implementa con implements.

```
class Imprimible {
  void imprimir();
}

class Documento implements Imprimible {
```

```
  @override
  void imprimir() => print("Imprimiendo documento");
}
```

🔷 **Clases abstractas**

- No se pueden instanciar.
- Sirven como base para otras clases.

```
abstract class Figura {
  double area();
}

class Circulo extends Figura {
  double radio;
  Circulo(this.radio);
  @override
  double area() => 3.1416 * radio * radio;
}
```

💡 **Resumen POO en Dart**

- Clase: define atributos y métodos.
- Constructor: inicializa objetos.
- Factory: control sobre creación de instancias.
- Herencia: extends para especializar.
- Mixins: with para añadir comportamientos.
- Interfaces implícitas: se implementan con implements.
- Clases abstractas: para definir plantillas de comportamiento.

7. Features Avanzados

7.1. Extensions

🔷 **¿Qué son?**

Las *extensions* permiten añadir nuevos métodos o propiedades a clases ya existentes **sin modificarlas**.

📌 **Ejemplo**:

dart

Copiar código

```
extension StringMayus on String {
  String primeraMayus() {
```

```
    if (isEmpty) return this;
    return this[0].toUpperCase() + substring(1);
  }
}

void main() {
  print("hola mundo".primeraMayus()); // Hola mundo
}
```

💡 **Ventaja**:
Puedes añadir utilidades personalizadas a tipos nativos o de librerías externas.

7.2. Generators

🔷 **Concepto**
Un *generator* produce elementos **bajo demanda** (lazy evaluation).

- `sync*` → genera secuencias síncronas.
- `async*` → genera secuencias asíncronas.

📌 **Ejemplo con `sync*`**:

dart
Copiar código

```
Iterable<int> contarHasta(int max) sync* {
  for (int i = 1; i <= max; i++) {
    yield i; // devuelve un valor y "pausa" la función
  }
}

void main() {
  for (var n in contarHasta(5)) {
    print(n); // 1 2 3 4 5
  }
}
```

📌 **Ejemplo con `async*`**:

dart

Copiar código

```
Stream<int> contarAsync(int max) async* {
  for (int i = 1; i <= max; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i;
  }
}

void main() async {
  await for (var n in contarAsync(3)) {
    print(n); // imprime un número por segundo
  }
}
```

💬 **Usos comunes**:

- Lectura progresiva de datos.
- Streams de eventos.
- Generación de listas grandes sin cargar todo en memoria.

7.3. Callable Classes

🔷 **¿Qué son?**
Permiten que un objeto se **use como si fuera una función** implementando el método especial `call()`.

📌 **Ejemplo**:

dart

Copiar código

```
class Suma {
  int call(int a, int b) => a + b;
}

void main() {
  var sumar = Suma();
  print(sumar(3, 4)); // 7
}
```

💡 **Ventaja**:
Útil para objetos que representan operaciones o servicios, haciéndolos más naturales de usar.

📖 **Resumen del Punto 7**

- **Extensions** → Añadir métodos a clases existentes.
- **Generators** → Crear secuencias *lazy* con `sync*` o `async*`.
- **Callable Classes** → Objetos que actúan como funciones.

8. Asincronía y Concurrencia

8.1. Futures y async/await

🔷 **¿Qué es un `Future`?**

- Representa un valor que estará disponible **en el futuro**.
- Puede completarse con éxito (valor) o con error (excepción).

📌 **Ejemplo básico**:

dart
Copiar código

```
Future<String> obtenerDatos() async {
  await Future.delayed(Duration(seconds: 2)); // simula espera
  return "Datos recibidos";
}

void main() async {
  print("Cargando...");
  var datos = await obtenerDatos();
  print(datos); // Datos recibidos
}
```

🔷 **Manejo de errores con `try/catch`**

dart
Copiar código

```
Future<void> cargarArchivo() async {
  try {
```

```
    await Future.delayed(Duration(seconds: 1));
    throw Exception("Archivo no encontrado");
  } catch (e) {
    print("Error: $e");
  }
}

void main() async {
  await cargarArchivo();
}
```

💡 **Tip**:
Usa `await` para escribir código asíncrono como si fuera síncrono, mejorando la legibilidad.

8.2. Streams

🔷 **¿Qué es un `Stream`?**

- Secuencia de datos que llega **con el tiempo**.
- Puede emitir múltiples valores y finalizar o emitir un error.

📌 **Ejemplo básico**:

dart
Copiar código

```
Stream<int> contadorStream(int max) async* {
  for (int i = 1; i <= max; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i;
  }
}

void main() async {
  await for (var n in contadorStream(3)) {
    print(n); // 1, 2, 3 (un número por segundo)
  }
}
```

🔷 Uso con `StreamController`

dart
Copiar código

```
import 'dart:async';

void main() {
  final controller = StreamController<String>();

  // Suscriptor
  controller.stream.listen((dato) {
    print("Recibido: $dato");
  });

  // Emisión de datos
  controller.add("Hola");
  controller.add("Mundo");

  controller.close();
}
```

💬 **Usos comunes**:

- Lectura de sensores.
- Eventos de usuario.
- Comunicación en tiempo real.

8.3. Isolates

🔷 Concepto básico

- Un *isolate* es como un hilo independiente con su propia memoria.
- Evita bloqueos en operaciones intensivas de CPU.
- Comunicación mediante *mensajes*.

📌 **Ejemplo simple**:

dart
Copiar código

```
import 'dart:isolate';

void tareaPesada(SendPort sendPort) {
  var resultado = 0;
  for (var i = 0; i < 100000000; i++) {
    resultado += i;
  }
  sendPort.send(resultado);
}

void main() async {
  final receivePort = ReceivePort();

  await Isolate.spawn(tareaPesada, receivePort.sendPort);

  receivePort.listen((mensaje) {
    print("Resultado: $mensaje");
    receivePort.close();
  });
}
```

💡 **Tip**:
Usa isolates solo cuando realmente haya tareas **CPU-bound** que puedan bloquear la interfaz.

📖 **Resumen del Punto 8**

- **Futures** → Ejecutar tareas que toman tiempo y obtener el resultado más tarde.
- **async/await** → Sintaxis simplificada para trabajar con futures.
- **Streams** → Recibir múltiples valores a lo largo del tiempo.
- **Isolates** → Procesos ligeros independientes para trabajo intensivo.

9. Buenas Prácticas en Dart

Trabajar con Dart no es solo conocer la sintaxis: la calidad del código, su legibilidad y

mantenibilidad son esenciales para proyectos reales, especialmente si se desarrollan con Flutter o en entornos colaborativos.

A continuación se recogen **buenas prácticas clave** para programar en Dart de forma profesional.

9.1. Nombres y Convenciones

🔷 **Estilo de nombres oficial (Effective Dart)**

Variables y funciones → *lowerCamelCase*

dart
Copiar código
```
var nombreUsuario = "Ana";

void enviarMensaje() {}
```

-

Clases y enumeraciones → *UpperCamelCase*

dart
Copiar código
```
class Usuario {}

enum EstadoPedido { pendiente, enviado }
```

-

Constantes → *lowerCamelCase* (no usar mayúsculas como en C).

dart
Copiar código
```
const maxIntentos = 3;
```

-

Privacidad → Anteponer `_` para indicar miembro privado a la librería.

dart
Copiar código
```
String _clavePrivada = "1234";
```

-

📖 **Importante**:
Seguir una convención consistente mejora la colaboración y evita confusiones.

9.2. Código Limpio y Legible

🔷 **Regla de oro**:

> "El código se lee más veces de las que se escribe."

💡 **Recomendaciones**:

1. **Funciones cortas**: una función debería hacer una sola cosa.

Evitar "números mágicos": usar constantes con nombres descriptivos.

dart
Copiar código
```
const segundosTimeout = 30; // mejor que usar "30" directamente
```

2.

Comentarios claros: explicar el *por qué*, no el *qué*.

dart
Copiar código
```
// Usamos este método para mejorar la velocidad de búsqueda en
listas grandes
```

3.
4. **Agrupar código relacionado**: mantener juntas variables y métodos que tengan relación.

9.3. Uso de Null Safety Correcto

Dart tiene null safety integrada, pero **forzarlo incorrectamente con `!` puede provocar fallos**.

📌 **Buenas prácticas**:

- Preferir `?.` y `??` antes que `!`.
- Inicializar variables en el constructor cuando sea posible.
- Usar `late` solo cuando sea realmente necesario y seguro.

❌ **Mala práctica**:

dart

Copiar código

```
String? nombre;
print(nombre!.length); // puede lanzar error
```

✅ **Buena práctica**:

dart

Copiar código

```
String? nombre;
print(nombre?.length ?? 0);
```

9.4. Organización de Archivos y Paquetes

📦 **Estructura recomendada para proyectos medianos/grandes**:

bash

Copiar código

```
lib/
 ├─ models/        # Clases de datos (POJOs)
 ├─ services/      # Lógica de negocio y APIs
 ├─ utils/         # Funciones de ayuda
 ├─ widgets/       # Componentes reutilizables (Flutter)
 └─ main.dart      # Punto de entrada
```

💡 **Tip**:

- Un archivo no debería tener más de una clase pública principal.
- Evitar archivos con cientos de líneas: dividir por responsabilidad.

9.5. Documentación Integrada

🔷 **Comentarios de documentación `///`**
Estos comentarios son procesados por herramientas como `dart doc`.

📌 **Ejemplo**:

dart

Copiar código

```
/// Clase que representa un usuario en el sistema.
/// [nombre] es obligatorio y no puede ser vacío.
class Usuario {
  final String nombre;
  Usuario(this.nombre);
}
```

Ventajas:

- Genera documentación HTML automáticamente.
- Mejora la comprensión del código para otros desarrolladores.

9.6. Análisis de Código y Linting

📖 **Linting**: conjunto de reglas automáticas para mantener un estilo uniforme y detectar errores potenciales.

Pasos recomendados:

1. Crear un archivo `analysis_options.yaml` en la raíz del proyecto.
2. Activar reglas recomendadas:

yaml

Copiar código

```
include: package:flutter_lints/flutter.yaml
```

```
linter:
  rules:
    avoid_print: true
    prefer_const_constructors: true
    always_declare_return_types: true
```

3. Ejecutar el análisis:

bash

Copiar código

```
dart analyze
```

💡 **Tip**:
Configura el análisis como parte del proceso de CI/CD para que el código siempre pase la revisión automática antes de integrarse.

9.7. Buen Uso de Colecciones

- Preferir métodos funcionales (`map`, `where`, `fold`, `reduce`) frente a bucles manuales cuando mejore la legibilidad.
- Usar `const []`, `const {}` cuando las colecciones sean inmutables para optimizar rendimiento.

📌 **Ejemplo**:

dart

Copiar código

```
final lista = const [1, 2, 3];
final cuadrados = lista.map((n) => n * n).toList();
```

9.8. Manejo de Errores

- Usar `try/catch` para capturar excepciones y actuar en consecuencia.
- Añadir `on` para capturar excepciones específicas.

dart

Copiar código

```
try {
  var resultado = 10 ~/ 0;
} on IntegerDivisionByZeroException {
  print("División entre cero no permitida");
} catch (e) {
  print("Error inesperado: $e");
}
```

- Evitar capturar errores sin hacer nada con ellos.

9.9. Optimización y Buen Rendimiento

- Usar `const` para widgets inmutables (en Flutter).
- Evitar cálculos pesados en el método `build()` de Flutter; delegarlos a variables o funciones externas.
- Reutilizar objetos en lugar de crearlos repetidamente.

9.10. Principios Generales de Diseño

1. **DRY** (*Don't Repeat Yourself*) → evitar duplicación de código.
2. **KISS** (*Keep It Simple, Stupid*) → soluciones simples siempre que sea posible.
3. **YAGNI** (*You Aren't Gonna Need It*) → no implementar cosas "por si acaso" que no se usan.

4. **SOLID** → principios de diseño orientado a objetos para mantenibilidad.

📖 **Resumen del Punto 9 (Buenas Prácticas)**

- Nombrado y estilo coherente → facilita trabajo en equipo.
- Funciones cortas y legibles → evitan complejidad innecesaria.
- Null safety usado con precaución.
- Organización clara en carpetas y archivos.
- Documentar con `///` y mantener reglas de linting.
- Manejar errores adecuadamente.
- Pensar en optimización y rendimiento desde el inicio.

10. Documentar Software en Dart

La documentación es una parte fundamental del desarrollo profesional: **un buen código sin documentación es como un mapa sin leyenda**. Aunque Dart sea un lenguaje de sintaxis clara, siempre habrá partes del código que necesiten explicación, ya sea para otros desarrolladores o para ti mismo en el futuro.

10.1. ¿Por qué documentar?

🔷 **Razones principales**:

1. **Mantenimiento** → Facilita que el código pueda actualizarse sin romper funcionalidades.
2. **Colaboración** → Otros desarrolladores pueden entender más rápido la lógica y la intención del código.
3. **Autoaprendizaje** → Documentar te obliga a pensar mejor cómo y por qué haces algo.
4. **Generación automática de documentación HTML** con `dart doc`, útil para distribuir APIs internas o librerías públicas.
5. **Soporte a largo plazo** → Si vuelves a un proyecto meses después, la documentación será tu memoria escrita.

10.2. Tipos de comentarios en Dart

En Dart existen tres formas de comentar, cada una con un propósito específico:

Tipo	Símbolo	Uso principal
Comentario de una línea	`//`	Notas rápidas o explicación puntual
Comentario multilínea	`/* ... */`	Bloques grandes de texto o descripciones temporales
Comentario de documentación	`///`	Documentación formal procesada por `dart doc`

10.2.1. Comentarios de documentación (`///`)

Estos comentarios:

- Se colocan **justo encima** de la declaración (clase, método, variable pública).
- Son procesados por herramientas como `dart doc` para generar documentación HTML.
- Pueden usar referencias entre corchetes `[ ]` para enlazar a otras clases, métodos o variables.

📌 **Ejemplo básico**:

dart

Copiar código

```
/// Clase que representa un usuario en el sistema.
/// Contiene información básica como [nombre] y [edad].
class Usuario {
  /// Nombre completo del usuario.
  final String nombre;

  /// Edad del usuario en años.
```

```
  final int edad;

  /// Crea un nuevo usuario con [nombre] y [edad].
  Usuario(this.nombre, this.edad);

  /// Saluda al usuario en la consola.
  void saludar() => print("Hola, soy $nombre");
}
```

💡 **Regla**: Documenta **clases**, **métodos públicos** y **parámetros importantes**.

Ejemplo con parámetro documentado:

dart

Copiar código

```
/// Calcula el área de un rectángulo.
///
/// [ancho] y [alto] deben ser positivos.
double areaRectangulo(double ancho, double alto) => ancho * alto;
```

10.2.2. Comentarios en línea (`//`)

- Úsalos para aclarar partes concretas del código que podrían resultar confusas.
- No describas lo obvio: evita comentarios que repiten exactamente lo que hace el código.

❌ **Ejemplo innecesario**:

dart

Copiar código

```
// Incrementa el contador en 1
contador++;
```

✅ **Ejemplo útil**:

dart

Copiar código

```
// Evitamos que el contador supere el valor máximo permitido
if (contador < maxContador) contador++;
```

10.2.3. Comentarios multilínea (`/* ... */`)

- Útiles para documentación temporal o deshabilitar bloques grandes de código durante depuración.
- No deben usarse como forma habitual de documentar API pública (para eso está `///`).

📌 **Ejemplo**:

dart

Copiar código

```
/*
Este bloque de código implementa una solución alternativa
para sistemas que no soportan la API principal.
Se mantendrá hasta que la migración esté completada.
*/
```

10.3. Generar documentación automática

1. Asegúrate de que las clases, métodos y propiedades públicas tienen comentarios `///`.
2. Ejecuta en la terminal:

bash

Copiar código

```
dart doc
```

3. La documentación se generará en la carpeta `doc/api` en formato HTML.

💡 **Consejo**:

- Si desarrollas librerías para terceros, mantener esta documentación actualizada es tan importante como el código mismo.

10.4. Buenas prácticas al documentar en Dart

1. **Escribir en presente** → "Devuelve la suma…" en lugar de "Devuelve la suma que…"
2. **Ser conciso pero claro** → No escribir párrafos innecesarios.

Usar ejemplos en la documentación para métodos complejos:

dart
Copiar código

```
/// Devuelve el cuadrado de un número.
///
/// Ejemplo:
/// ```dart
/// var resultado = cuadrado(4); // 16
/// ```
int cuadrado(int x) => x * x;
```

3.
4. **Mantener sincronizada la documentación** cuando se actualiza el código.
5. **No documentar código obvio** — El mejor código es aquel que se entiende con la menor cantidad de comentarios posible.

10.5. Errores comunes al documentar

❌ Copiar y pegar la misma descripción en varios métodos.
❌ Dejar comentarios obsoletos que ya no corresponden al código actual.
❌ Usar comentarios para explicar errores en lugar de corregirlos.

✅ Documentar la intención y el propósito del código.
✅ Actualizar siempre junto con la lógica.

10.6. Ejemplo completo con buenas prácticas

dart

Copiar código

```dart
/// Servicio para gestionar pedidos en el sistema.
///
/// Este servicio permite crear, actualizar y cancelar pedidos.
/// Ejemplo de uso:
/// ```dart
/// var servicio = ServicioPedidos();
/// var pedido = servicio.crearPedido("Producto A", 3);
/// servicio.cancelarPedido(pedido.id);
/// ```
class ServicioPedidos {
  /// Crea un nuevo pedido con [producto] y [cantidad].
  Pedido crearPedido(String producto, int cantidad) {
    // TODO: Implementar conexión con la base de datos
    return Pedido(producto, cantidad);
  }

  /// Cancela un pedido dado su [id].
  void cancelarPedido(int id) {
    print("Pedido $id cancelado");
```

```
  }
}

class Pedido {
  final String producto;
  final int cantidad;
  final int id = DateTime.now().millisecondsSinceEpoch;

  Pedido(this.producto, this.cantidad);
}
```

11. Crear Juegos de Prueba en Dart

En Dart, los juegos de prueba (test suites) permiten verificar que el código funciona como se espera, detectar errores antes de que lleguen a producción y mantener la calidad a largo plazo.

11.1. Tipos de pruebas en Dart

- **Unit tests**
  Verifican una función o clase de forma aislada. Son rápidos y no dependen de recursos externos.

- **Pruebas de integración (Dart)**
  Comprueban cómo interactúan varias partes del código (sin interfaz gráfica).

- **Pruebas de rendimiento**
  Evalúan cuánto tarda un algoritmo o función en ejecutarse.

💡 En Dart no existe la distinción *widget test* o *golden test* como en Flutter; las pruebas suelen ser unitarias o de integración.

11.2. Configuración del entorno de pruebas

En el archivo `pubspec.yaml`, añade la dependencia de desarrollo:

yaml
Copiar código
```yaml
dev_dependencies:
  test: ^1.25.0
```

Instala dependencias:

bash
Copiar código
```bash
dart pub get
```

11.3. Estructura recomendada

bash
Copiar código
```
mi_proyecto/
 ├─ lib/           # Código fuente principal
 │   └─ calculadora.dart
 └─ test/          # Archivos de prueba
     └─ calculadora_test.dart
```

📌 **Regla**: Los archivos de prueba deben terminar en `_test.dart` para que `dart test` los reconozca automáticamente.

11.4. Escribir pruebas unitarias

Ejemplo: código a probar (`lib/calculadora.dart`):

dart
Copiar código
```dart
int sumar(int a, int b) => a + b;

double dividir(int a, int b) {
  if (b == 0) throw ArgumentError('División por cero');
  return a / b;
}
```

Ejemplo: pruebas (`test/calculadora_test.dart`):

dart
Copiar código

```dart
import 'package:test/test.dart';
import '../lib/calculadora.dart';

void main() {
  group('Pruebas de la calculadora', () {
    test('Suma correcta', () {
      expect(sumar(2, 3), equals(5));
    });

    test('División correcta', () {
      expect(dividir(6, 2), equals(3));
    });

    test('División por cero lanza error', () {
      expect(() => dividir(4, 0), throwsA(isA<ArgumentError>()));
    });
  });
}
```

Ejecutar:

bash
Copiar código

```bash
dart test
```

11.5. Pruebas con código asíncrono

En Dart, las funciones que devuelven `Future` deben probarse con `async` y `await`.

Ejemplo:

dart
Copiar código

```dart
Future<String> obtenerDatos() async {
  await Future.delayed(Duration(milliseconds: 100));
  return "Datos recibidos";
}
```

```
void main() {
  test('obtenerDatos devuelve el mensaje esperado', () async {
    final resultado = await obtenerDatos();
    expect(resultado, equals("Datos recibidos"));
  });
}
```

11.6. Probar Streams

Un `Stream` puede emitir varios valores en el tiempo, por lo que se prueba con `emits`, `emitsInOrder` o `expectLater`.

Ejemplo:

dart

Copiar código

```
import 'dart:async';
import 'package:test/test.dart';

Stream<int> contador(int hasta) async* {
  for (var i = 1; i <= hasta; i++) {
    yield i;
  }
}

void main() {
  test('contador emite valores en orden', () {
    expect(contador(3), emitsInOrder([1, 2, 3, emitsDone]));
  });
}
```

11.7. Agrupación y organización con `group()`

`group()` permite agrupar tests relacionados para mantenerlos organizados.

dart

Copiar código

```
void main() {
  group('Operaciones matemáticas', () {
```

```
    test('Suma', () {
      expect(2 + 2, equals(4));
    });

    test('Resta', () {
      expect(5 - 3, equals(2));
    });
  });
}
```

11.8. Uso de `setUp()` y `tearDown()`

- `setUp()` → se ejecuta antes de cada test.
- `tearDown()` → se ejecuta después de cada test.

Ejemplo:

dart
Copiar código

```
int contador = 0;

void main() {
  setUp(() {
    contador = 0; // reiniciar antes de cada prueba
  });

  test('incrementa contador', () {
    contador++;
    expect(contador, equals(1));
  });

  tearDown(() {
    // Aquí podrías cerrar conexiones o limpiar datos temporales
  });
}
```

11.9. Pruebas con datos simulados (Mocking)

Para evitar dependencias reales (como red o base de datos), se usan clases simuladas.

Ejemplo:

dart
Copiar código

```dart
abstract class Api {
  Future<String> fetchData();
}

class ApiFake implements Api {
  @override
  Future<String> fetchData() async => "Respuesta simulada";
}

void main() {
  test('usa ApiFake para pruebas', () async {
    final api = ApiFake();
    final datos = await api.fetchData();
    expect(datos, equals("Respuesta simulada"));
  });
}
```

💡 Esto evita que las pruebas dependan de internet o APIs externas.

11.10. Medir cobertura de pruebas

Para ver qué porcentaje del código está cubierto por tests:

bash
Copiar código

```bash
dart test --coverage=coverage
```

Después, puedes generar un reporte legible con `lcov`:

bash
Copiar código

```bash
dart pub global activate coverage
dart pub global run coverage:format_coverage \
           --lcov      --in=coverage      --out=coverage/lcov.info
--packages=.packages
```

11.11. Buenas prácticas en pruebas Dart

1. **Aislar pruebas**: que no dependan del estado de otras.
2. **Nombrar bien**: el nombre del test debe indicar claramente qué verifica.
3. **Cubrir casos normales y extremos**: entradas válidas, límites y errores esperados.
4. **Evitar sleeps innecesarios**: usar APIs de testing para manejar asincronía.
5. **Ejecutar las pruebas antes de cada commit** para detectar errores a tiempo.

📖 **Resumen del Punto 11**

- Usa `package:test` para escribir y ejecutar pruebas en Dart.
- Agrupa y organiza las pruebas con `group()`, `setUp()` y `tearDown()`.
- Maneja correctamente asincronía con `async`/`await` y Streams con `emitsInOrder`.
- Simula dependencias externas con clases fake o mocks.
- Mide la cobertura y mantén las pruebas actualizadas.

12. Recursos para aprender y trabajar con Dart

Dart cuenta con una amplia comunidad y recursos oficiales y no oficiales que facilitan su aprendizaje y aplicación profesional. Aquí tienes una selección organizada por tipo de recurso.

12.1. Documentación oficial y guías

- **Página oficial de Dart**
  Sitio central con descargas, documentación y noticias sobre el lenguaje.
  URL: https://dart.dev

- **Guía de lenguaje Dart**
  Explicación completa de sintaxis, tipos, null safety y funcionalidades avanzadas.
  URL: https://dart.dev/guides/language

- **API de Dart**

Referencia de todas las clases y funciones incluidas en el SDK.
URL: https://api.dart.dev

- **Guía de Null Safety**
  Explicación de cómo funciona la seguridad contra valores nulos en Dart.
  URL: https://dart.dev/null-safety

12.2. Tutoriales y cursos interactivos

- **DartPad** (editor online)
  Permite escribir, ejecutar y compartir código Dart directamente en el navegador.
  URL: https://dartpad.dev

- **Tour of the Dart Language**
  Recorrido interactivo por las principales características del lenguaje.
  URL: https://dart.dev/guides/language/language-tour

- **Ejercicios de práctica en Dart**
  Colección de retos para practicar programación en Dart.
  URL: https://dart.dev/codelabs

12.3. Herramientas y utilidades

- **Dart SDK**
  Descarga del SDK oficial, necesario para ejecutar y compilar programas Dart.
  URL: https://dart.dev/get-dart

- **Paquetes y librerías en pub.dev**
  Repositorio oficial de paquetes de Dart y Flutter, con buscador y documentación.
  URL: https://pub.dev

- **Dart Analyzer**
  Herramienta para revisar y mantener calidad de código, integrada en el SDK.
  URL: https://dart.dev/tools/dart-analyze

- **Dart Formatter**
  Formatea automáticamente el código siguiendo las guías oficiales de estilo.
  URL: https://dart.dev/tools/dart-format

12.4. Comunidad y soporte

- **Foro oficial de Dart** (Dart Discussions)
  Espacio para preguntas, debates y anuncios.
  URL: https://github.com/dart-lang/sdk/discussions

- **Stack Overflow (etiqueta Dart)**
  Gran base de preguntas y respuestas técnicas sobre Dart.
  URL: https://stackoverflow.com/questions/tagged/dart

- **Twitter/X oficial de Dart**
  Noticias, actualizaciones y recursos publicados por el equipo oficial.
  URL: https://twitter.com/dart_lang

12.5. Blogs y artículos recomendados

- **Medium - Dart Language**
  Recopilación de artículos y tutoriales de la comunidad.
  URL: https://medium.com/tag/dart

- **Dart en dev.to**
  Publicaciones técnicas y ejemplos prácticos.
  URL: https://dev.to/t/dart

📌 **Consejo final**:
El mejor aprendizaje combina **lectura de documentación oficial** + **práctica en DartPad** + **participación en la comunidad** para resolver dudas y compartir código.