Walter Alexander Mata López
Mónica Cobián Alvarado
Armando Román Gallardo
José Román Herrera Morales

Programación estructurada

y orientada a objetos con Dart

Universidad de Colima

Programación estructurada

y orientada a objetos con Dart

En buen plan

Programación estructurada

y orientada a objetos con Dart

Walter Alexander Mata López
Mónica Cobián Alvarado
Armando Román Gallardo
José Román Herrera Morales

Universidad de Colima

Programación estructurada y orientada a objetos con Dart

Avenida Universidad 333
Colima, Colima, México
Dirección General de Publicaciones
Teléfonos: 312 316 1081 y 312 316 1000, ext. 35004
Correo electrónico: publicaciones@ucol.mx
www.ucol.mx

Publicado en México / *Published in Mexico*

ISBN electrónico: 978-968-9733-20-1
DOI: 10.53897/LI.2026.0004.UCOL
5E.1.1/317010/423/2025 Edición de publicación no periódica

Proceso editorial certificado con normas ISO desde 2005
Dictaminación doble ciego y edición registrada en el Sistema Editorial Electrónico Pred

Registro: CU-022-25
Recibido: Octubre de 2025
Publicado: Febrero de 2026

Índice

Prólogo

Es un placer presentar *Programación estructurada y orientada a objetos con Dart*, una obra diseñada especialmente para docentes y estudiantes de las carreras de ingeniería en la Universidad de Colima y otros centros educativos similares. Se trata de una herramienta fundamental para cualquier persona interesada en adquirir conocimientos sólidos en programación, especialmente en el contexto del lenguaje Dart.

La programación es una habilidad esencial en la era digital en la que vivimos. Cada vez más, las industrias y empresas de todo tipo requieren programadores y desarrolladores de software altamente capacitados. El libro tiene como objetivo proporcionar una base sólida en programación utilizando Dart, un lenguaje de programación moderno y versátil que se ha convertido en una elección popular para el desarrollo de aplicaciones web y móviles.

A lo largo de estas páginas, las y los estudiantes encontrarán un enfoque claro y estructurado para aprender los fundamentos de la programación, desde conceptos básicos hasta temas más avanzados con Dart. El libro se organiza en dos secciones principales:

- **Parte I. Programación estructurada con Dart** presenta los conceptos iniciales, incluida una introducción a Dart, la instalación del entorno de desarrollo, tipos de datos, estructuras de control y muchas otras bases fundamentales para programar en Dart.
- **Parte II. Programación orientada a objetos con Dart** se adentra en la programación orientada a objetos, un paradigma crucial para el desarrollo de aplicaciones escalables y eficientes. Aquí, las y los estudiantes aprenderán sobre clases, herencia, polimorfismo y otros conceptos clave de la POO.

También se incluyen algunos apartados dedicados a temas avanzados, como programación asíncrona, manejo de errores y excepciones, colecciones, funciones y genéricos, para garantizar que el estudiantado tenga una comprensión completa de Dart y de la programación en general.

Este texto se basa en la premisa de que la programación es una habilidad que se aprende mejor con la práctica. Por lo tanto, se incluyen ejemplos de código práctico en cada capítulo, junto con ejercicios y desafíos para reforzar el aprendizaje. Para poner en práctica lo aprendido, se puede utilizar Dartpad o una instalación local de Dart.

Diseñado para materias relacionadas con la programación, abarca desde los conceptos básicos hasta las áreas más avanzadas de programación en Dart. El objetivo es proporcionar a las y los docentes y estudiantes una herramienta valiosa para adquirir habilidades que serán útiles en su carrera y en el mundo tecnológico en constante evolución.

Estamos seguros de que este libro les brindará una base sólida y una comprensión profunda de la programación en Dart, preparándolos para enfrentar desafíos en sus futuros proyectos.

Walter Alexander Mata López
Mónica Cobián Alvarado
Armando Román Gallardo
José Román Herrera Morales

Parte I
Programación estructurada con Dart

En esta sección exploraremos los fundamentos de la programación estructurada utilizando el lenguaje de programación Dart. Abordaremos una amplia gama de temas, desde una introducción a la programación con Dart hasta conceptos más avanzados como la manipulación de colecciones y la programación asíncrona. Comenzaremos con una breve historia y características de Dart a manera de introducción, luego, nos sumergiremos en los conceptos clave, como variables, tipos de datos, operadores y estructuras condicionales. También exploraremos cómo trabajar con colecciones, funciones y manejo de errores.

A lo largo de esta sección, te guiaremos a través de ejemplos prácticos, para que puedas comprender y aplicar estos conceptos de manera efectiva en tus proyectos. Desde la sintaxis básica hasta técnicas más avanzadas, a lo largo de varios capítulos te proporcionaremos ejemplos y ejercicios para que puedas abordar proyectos de programación estructurada utilizando Dart.

Capítulo 1
Introducción a la programación con Dart

En este capítulo nos adentraremos en el mundo de la programación con Dart, un lenguaje de programación moderno y versátil desarrollado por Google. Abordaremos aspectos esenciales que servirán como cimientos para tu comprensión y uso efectivo de Dart en tus proyectos. Exploraremos las características distintivas de Dart que lo hacen sobresalir en el panorama de la programación y analizaremos las posibilidades que este lenguaje ofrece en términos de desarrollo.

Además, revisaremos los entornos de desarrollo disponibles, incluyendo herramientas como Dartpad y cómo realizar la instalación local para tener un entorno de desarrollo configurado. Aprenderemos cómo crear programas y proyectos nuevos en Dart, así como cómo estructurar nuestro código de manera efectiva. También abordaremos la importancia de los comentarios en el código y cómo acceder a la documentación oficial de Dart para potenciar nuestra comprensión y uso de este poderoso lenguaje.

Finalmente, exploraremos las palabras reservadas en Dart y cómo influyen en la sintaxis y estructura de nuestros programas.

Breve historia de Dart

Como ya mencionamos anteriormente, Dart es un lenguaje de programación moderno desarrollado por Google, que ha generado bastantes expectativas y ha ganado la atención de la comunidad de desarrollo de software. Al ser de código abierto, Dart cuenta con una comunidad que contribuye a su ecosistema de manera activa. Desde su lanzamiento inicial en el 2013 con la versión 1.0, Dart ha experimentado un desarrollo y crecimiento significativos.

En sus primeras etapas, Dart utilizaba una herramienta llamada Dart2js para transpilar su código a JavaScript, lo que permitía ejecutar aplicaciones Dart en navegadores web que no soportaban el lenguaje de forma nativa; sin embargo, con el lanzamiento de la Máquina Virtual de Dart en su versión 1.9 se empezó a vislumbrar el potencial de la tecnología.

Un cambio particularmente notorio vino con el Proyecto Sky, un prototipo inicial para el desarrollo de aplicaciones móviles que más tarde se transformaría en Flutter en el año 2017, el cual se convirtió rápidamente en una de los frameworks más populares para el desarrollo de aplicaciones móviles, impulsado por la eficiencia y versatilidad de Dart como su lenguaje de programación subyacente.

La versión 2.0 de Dart, lanzada en 2018 fue muy importante, pues se llevó a cabo de manera simultánea con el lanzamiento de la beta de Flutter. La entrega de Dart 2.6 en 2021, expandió las capacidades del lenguaje para el desarrollo no sólo de aplicaciones móviles, sino también de aplicaciones web y de escritorio. La versión 3.0 de Flutter, que

acompaña a Dart 2.6, añade capacidades para desarrollar aplicaciones cruzadas que pueden ejecutarse en múltiples plataformas con el mismo código base.

Dart ha evolucionado de ser un lenguaje orientado principalmente a la web a convertirse en un lenguaje robusto y versátil, apto para el desarrollo de aplicaciones en múltiples plataformas. Su estrecha integración con Flutter lo coloca como un participante importante en el mundo del desarrollo móvil y multiplataforma. A continuación, podemos ver un esquema resumido de la historia de Dart.

- Es un lenguaje de programación desarrollado por Google
- Open Source
- En el 2013 se liberó la versión 1.0
  - Usaba Dart2js
  - Transpilaba a JS
- Salió la máquina virtual de Dart version 1.9
  - Proyecto Sky: prototipo para el desarrollo de aplicaciones móviles
  - En el 2017 cambia a Flutter
- Versión 2.0 (2018)
  - Se lanza la beta de Flutter
- Versión 2.6 (2021)
  - Desarrollo de aplicaciones web y de escritorio
  - Flutter 3.0
  - Aplicaciones cruzadas

Características de Dart

Dart es un lenguaje de programación orientado a objetos que destaca por su productividad, velocidad, portabilidad y accesibilidad. Su sintaxis es sencilla y simplificada, lo que facilita la escritura de código. En términos de rendimiento, Dart ofrece compilación anticipada y un alto rendimiento predecible, lo que lo hace rápido y eficiente. Otra de sus fortalezas es portabilidad; Dart compila a código ARM y x86 y ofrece dos tipos de compilación: Ahead-of-Time (AOT) y Just-in-Time (JIT). Esto, junto con la característica de recarga en caliente (hot reload), permite una gran flexibilidad y eficiencia en el desarrollo. Además, las aplicaciones escritas en Dart se ejecutan de forma nativa en una variedad de plataformas, incluidas Android, iOS, Linux, MacOS, Windows y la Web. Dart también es accesible para desarrolladores que ya tienen experiencia en lenguajes como C++, C# o Java, ya que su curva de aprendizaje es relativamente pequeña. Finalmente, el lenguaje es reactivo por naturaleza, permitiendo la programación asíncrona y la creación de objetos Futuros (Future) y Stream, lo que facilita la manipulación de operaciones que pueden requerir tiempo en completarse. Un breve resumen esquematizado de las características se muestra a continuación.

- Orientado a Objetos
- Productivo:
  - Sintaxis sencilla y simplificada
  - Herramientas simples pero poderosas
- Rápido:
  - Compilación anticipada
  - Alto rendimiento predecible
- Portable:
  - Compila a codigo ARM y x86
  - Tiene dos tipos de compilación: Ahead of time y Just in time
  - Hot reload
  - Se ejecutan de forma nativa en Android, iOS, Linux, Mac OS, Windows, Web.
- Accesible:
  - Curva de aprendizaje pequeña si se conoce C++, C# o Java
- Reactivo:
  - Asíncrono
  - Permite crear objetos Futuros (Future) y Stream

¿Qué se puede desarrollar con Dart?

Dart es un lenguaje de programación altamente versátil que se utiliza en diversos tipos de aplicaciones:

- Aplicaciones móviles: Principalmente a través de Flutter, Dart permite la compilación Ahead-of-Time (AoT) y Just-in-Time (JIT). Trabaja con una Máquina Virtual (VM) para lograr un rendimiento óptimo.
- Aplicaciones de escritorio: Además de Flutter, Dart Native también es una opción para el desarrollo de aplicaciones de escritorio. De igual forma, utiliza la compilación AoT, JIT y la VM.
- Servidores: Dart también se emplea en el desarrollo de servidores a través de frameworks como Angel, Aqueduct y Ktor, con soporte para multi-threading, lo que mejora la eficiencia en el manejo de múltiples tareas.
- Aplicaciones web: Flutter, Dart2js y Angular Dart son las principales herramientas para el desarrollo web en Dart. La transpilación, especialmente a través de Dart2js, permite que el código Dart se ejecute en navegadores web.

Como podemos apreciar, Dart ofrece un ecosistema robusto y flexible para el desarrollo de aplicaciones móviles, de escritorio, de servidor y web, apoyado por diferentes métodos de compilación y herramientas especializadas para cada tipo de aplicación.

Entornos de desarrollo

Dart ofrece una variedad de opciones tanto para desarrolladores novatos como para expertos en cuanto a la forma en que pueden empezar a trabajar con el lenguaje. Para aquellos que están dando sus primeros pasos en Dart o desean experimentar rápidamente con el lenguaje, el editor web DartPad es una excelente opción, permite escribir y ejecutar programas sencillos directamente desde el navegador, sin necesidad de instalación. Para los desarrolladores que requieren un entorno más robusto y completo, la instalación local es el camino para seguir. A continuación, se presentan algunas indicaciones para seguir cualquiera de estas dos opciones.

Dartpad

- Se pueden desarrollar programas sencillos en el editor web
- Ir al enlace: https://dartpad.dev/

Instalación local

- Instalación de Flutter
  - Ir a: https://docs.flutter.dev/get-started/install
  - Seleccionar la plataforma y seguir las instrucciones de instalación
- Instalación de Visual Studio Code
  - Ir a: https://code.visualstudio.com/Download
  - Seleccionar la plataforma y seguir las instrucciones de instalación
  - Instalar la extensión Dart en VSCode

Crear un nuevo programa de Dart

Para crear nuestro primer programa en Dart, podemos seguir las siguientes instrucciones:

- Crear un nuevo archivo en VSCode
- Escribir la extensión de archivo.dart
- Ejemplo: main.dart

Crear un nuevo proyecto de Dart

Otra forma de trabajar con Dart es mediante las opciones que tienen las distintas herramientas con las que dispone el ecosistema de Dart. Para crear un proyecto usamos la instrucción `dart` con la opción `create` como se indica a continuación.

- Abrir una Terminal (`cmd`) y escribir

```
C:\>dart create mi_app
```

- Aparecerá:

```
Creating mi_app using template console...

  .gitignore
  analysis_options.yaml
  CHANGELOG.md
  pubspec.yaml
  README.md
  bin\mi_app.dart
  lib\mi_app.dart
  test\mi_app_test.dart

Running pub get...
  Resolving dependencies...
  Downloading test 1.22.1...
  Downloading test_core 0.4.21...
  Downloading test_api 0.4.17...
  Downloading path 1.8.3...
  Downloading matcher 0.12.14...
  Downloading mime 1.0.3...
  Downloading frontend_server_client 3.2.0...
  Downloading analyzer 5.3.1...
  Downloading _fe_analyzer_shared 51.0.0...
  Changed 46 dependencies!

Created project mi_app in mi_app! In order to get started, run the following
commands:

  cd mi_app
  dart run
```

- Ingresamos al directorio mi_app

```
D:\wkspaceLearnDart>cd mi_app
```

- Ejecutamos el proyecto

```
D:\wkspaceLearnDart\mi_app>dart run
```

- Que tendrá como salida:

```
Building package executable...
Built mi_app:mi_app.
Hello world: 42!
```

Editar el código del proyecto

Una vez realizado lo anterior, podemos editar nuestro código, para esto vamos a usar el IDE VSCode, siguiendo las siguientes instrucciones:

- En la terminal y dentro del directorio del proyecto escribimos **`code .`**

```
D:\mi_app>code .
```

- Lo cual abrirá el Visual Studio Code que instalamos previamente.
- En la barra de navegación del proyecto (izquierda), dentro del directorio bin, se encuentra el archivo mi_app.dart

`Archivo: bin_app.dart`

```
import 'package:mi_app/mi_app.dart' as mi_app;

void main(List<String> arguments) {
  print('Hello world: ${mi_app.calculate()}!');
}
```

- El paquete `lib` contiene el archivo mi_app.dart con la función `calculate()`

```
int calculate() {
  return 6 * 7;
}
```

Estructura de un programa en Dart

```
void main() {
  print("Hola Mundo");
  print(1 + 3);
  print('Hello');
}
```

Comentarios

Dart ofrece diversas formas de insertar comentarios en el código, lo que facilita la documentación y la claridad de éste. Aquí hay una breve descripción de los diferentes tipos de comentarios que Dart soporta:

Comentarios de una sola línea

Los comentarios de una sola línea en Dart empiezan con `//`. Todo lo que sigue a estas dos barras inclinadas en la misma línea es considerado un comentario.

```
// Esto es un comentario en una sola línea
```

Comentario en varias líneas

Para comentarios que se extienden por más de una línea, Dart permite usar el estilo de comentario en bloque, que comienza con `/*` y termina con `*/`.

```
/*

Esto es un comentario
en un bloque que

Abarca varias líneas
*/
```

Comentarios de documentación

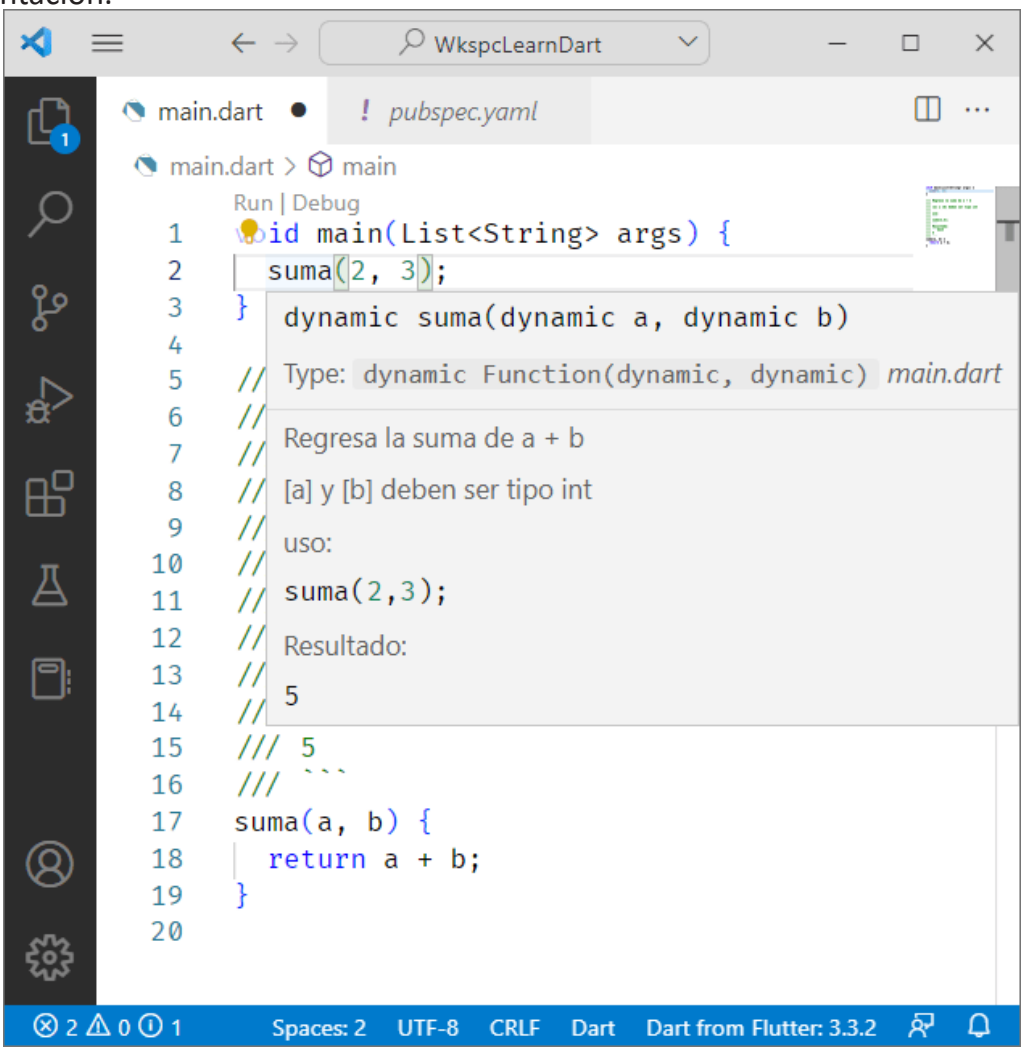
Dart también soporta comentarios de documentación, que son especialmente útiles para generar documentación automática para el código. Estos comentarios comienzan con tres barras inclinadas `///` en cada línea o con `/** ... */` para bloques de comentarios de documentación. Además, estos comentarios pueden incluir etiquetas Markdown y referencias especiales a variables, lo que ayuda a crear documentación más descriptiva y útil. A continuación, se presentan los comentarios de documentación de la función *suma*.

```
void main(List<String> args) {
  suma(2, 3);
}

/// Regresa la suma de a + b
///
/// [a] y [b] deben ser tipo int
///
/// uso:
/// ```
/// suma(2,3);
/// ```
/// Resultado:
/// ```bash
/// 5
/// ```
suma(a, b) {
  return a + b;
}
```

Como se puede observar, al hacer uso de la función `suma`, esta muestra la documentación que escribimos en el comentario. Es una ayuda considerable cuando tenemos muchas funciones, clases o métodos y necesitamos acceder de manera instantánea a su

documentación.



En este otro ejemplo se presenta una documentación de la clase `Operaciones` y las propiedades a y b para el método suma.

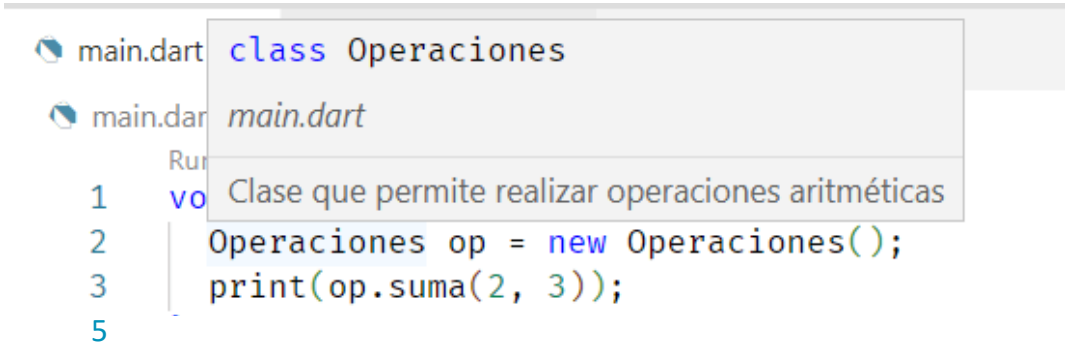
```
void main(List<String> args) {
  Operaciones op = new Operaciones();
  print(op.suma(2, 3));
}

///Clase que permite realizar operaciones aritméticas
class Operaciones {
  ///[a] es el primer argumento a sumar
  int? a;

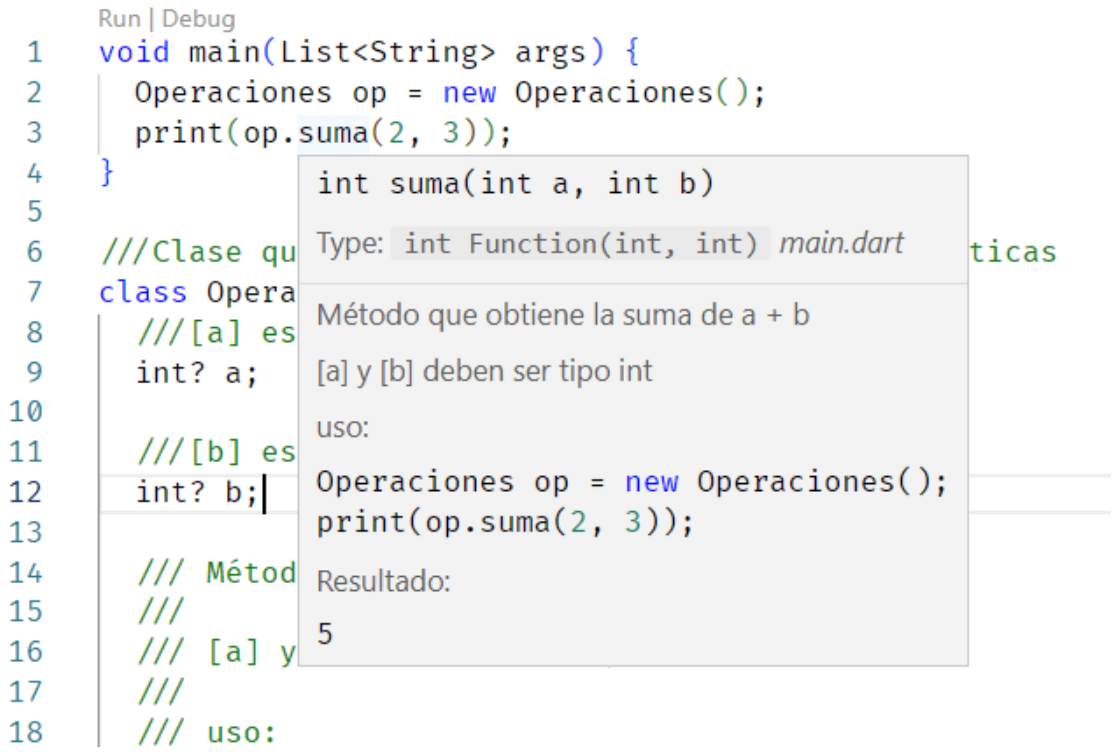
  ///[b] es el primer argumento a sumar
  int? b;
```

```
  /// Método que obtiene la suma de a + b
  ///
  /// [a] y [b] deben ser tipo int
  ///
  /// uso:
  /// ```
  /// Operaciones op = new Operaciones();
  /// print(op.suma(2, 3));
  /// ```
  /// Resultado:
  /// ```bash
  /// 5
  /// ```
  int suma(int a, int b) {
    return a + b;
  }
}
```

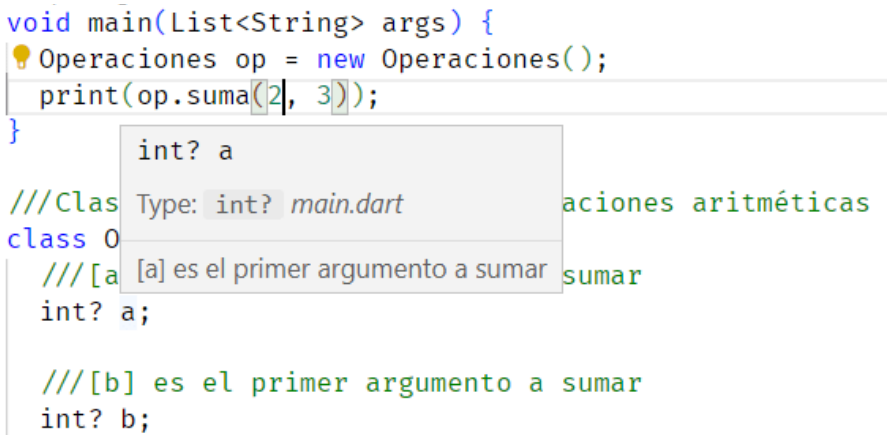
En esta pantalla se observa la documentación de la clase Operaciones en el IDE.



En esta pantalla se presenta cómo se ve en el IDE la documentación de los métodos de la clase `Operaciones`.



Ahora se presenta la documentación de las propiedades.



Como mencionamos al iniciar esta sección, una equivalencia a a /// son los bloques mediante `/**  …  */` como se muestra a continuación.

```
/**
*
*
*
*/
```

Acceder a la documentación de Dart

Para acceder a la documentación directa, en Windows se puede hacer control + click sobre el objeto o elemento del objeto para obtener la documentación.

Palabras reservadas

Las palabras reservadas en un lenguaje de programación son identificadores que tienen un significado predefinido y no pueden ser redefinidos por el usuario. Estas palabras son esenciales para la sintaxis y la estructura del lenguaje, y se utilizan para declarar variables, funciones, clases, propiedades, métodos, e instrucciones para controlar el flujo del programa, entre otras cosas. En el caso de Dart, el lenguaje cuenta con un conjunto de palabras reservadas que son fundamentales para escribir programas efectivos y comprensibles. Hasta el momento las palabras reservadas en Dart son:

abstract	deferred	for	part	try
as	do	get	required	typedef
assert	dynamic	if	rethrow	var
async	else	implements	return	void
await	enum	import	set	while
break	export	in	static	with
case	external	is	super	yield
catch	extends	late	switch	
class	factory	library	sync	
const	false	new	this	
continue	final	null	throw	
default	finally	operator	true	

Capítulo 2
Variables, tipos de datos y constantes

En este capítulo nos enfocamos en el mundo fundamental de las variables y tipos de datos en Dart. Estos elementos son esenciales para cualquier programador, ya que permiten almacenar y manipular información de diversas formas. A lo largo de este apartado exploraremos los conceptos clave, como el tipado explícito e inferencia de tipo, que nos ayudarán a comprender cómo Dart maneja los datos. Analizaremos en detalle los diferentes tipos de datos disponibles, centrándonos en cadenas de texto, números y lógicos. Aprenderemos a trabajar con estos tipos de datos, realizar operaciones aritméticas, utilizar operadores lógicos y aplicar técnicas de validación en nuestros programas. También abordaremos aspectos importantes como `null safety` y cómo manejar el dinamismo de los datos.

En Dart, el manejo de variables y tipos tiene algunas particularidades que son esenciales para entender cómo funciona el lenguaje. A continuación, se detallan estas características:

- Todos los tipos de datos son objetos: Esto significa que incluso los tipos primitivos como `int, double`, y `bool` son objetos. Esto permite que podamos invocar métodos sobre estos tipos, lo cual es una característica poco común en otros lenguajes de programación.
- Todos tienen por defecto el valor de *`null`*: Si no se asigna un valor a una variable en Dart, su valor por defecto será *`null`*. Esto es cierto para todos los tipos de datos, ya que, como mencionamos antes, todos son objetos.
- Tiene inferencia de tipo (default) con `var: Dart` es capaz de inferir el tipo de una variable en tiempo de compilación si se utiliza la palabra clave `var`. No obstante, una vez que se ha asignado un tipo a una variable, no se puede cambiar, por lo que hay que tener mucho cuidado al hacer uso de esta característica.
- Pueden iniciar con _ y `$:` Los nombres de variables en Dart pueden empezar con un guion bajo _ o un signo de dólar `$,` aunque esto tiene usos específicos. Los nombres que comienzan con un guion bajo son considerados identificadores privados en la biblioteca donde se definen.

```
El formato general para declarar una variable en Dart es <tipo_
de_dato> <nombre_variable> = <valor>;
```

Por ejemplo, en el siguiente código se declaran dos variables: *`_numeroEntero`* de tipo *`int`* con valor 10 y *`$numeroDouble`* de tipo *`double`* con valor 3.5. Como podemos observar, los nombres de las variables comienzan con _ y *`$,`* lo cual es válido en Dart. La siguiente línea suma las dos variables (*`_numeroEntero`* y *`$numeroDouble`*) mostrando el

resultado en la consola mediante la función *print*. Dado que una variable es de tipo *int* y la otra de tipo *double*, Dart realiza una conversión implícita para realizar la suma, dando como resultado un double. La salida del programa será 13.5, que es la suma de 10 y 3.5.

```
void main(List<String> args) {
  int _numeroEntero = 10;
  double $numeroDouble = 3.5;

  print(_numeroEntero + $numeroDouble);
}
```

Tipado explícito

Este tipado se refiere a la práctica de especificar el tipo de dato que una variable va a almacenar en el momento de su declaración; es decir, en lugar de utilizar la palabra clave var, que permite que Dart infiera el tipo de la variable, se indica el tipo deseado directamente. De manera general, podemos decir que:

- Se puede indicar siempre el tipo de dato antes del nombre de la variable

```
<tipo_de_dato> <nombre_variable> = <valor>;
```

- O asignar el valor posteriormente

```
<tipo_de_dato> <nombre_variable>;
<nombre_variable> = <valor>;
```

En el siguiente ejemplo se presentan los dos casos de tipado explícito, la variable *int a* adquiere el valor de 3 y a la variable *b* posteriormente se le asigna el valor de 5.

```
void main() {
  int a = 3;

  int b;
  b = 5;
}
```

Ventajas del tipado explícito

Este tipado tiene algunas ventajas, las cuales podemos resumir de la siguiente forma:

- Claridad: es más claro para los desarrolladores qué tipo de dato se espera.
- Prevención de errores: el compilador puede detectar errores de tipo más rápidamente, lo que facilita la depuración y mejora la calidad del código.
- Autocompletado y Ayuda: los IDEs modernos utilizan el tipo explícito para proporcionar funciones de autocompletado y ayuda en línea más precisas.
- Optimización de rendimiento: en algunos casos, el tipado explícito puede ayudar al compilador a realizar optimizaciones.

Tipado implícito

Para hacer uso del tipado implícito se usa la palabra clave `var` para declarar una variable sin especificar su tipo de dato. Dart infiere el tipo de la variable a partir del valor inicial asignado. Una vez que el tipo de la variable ha sido inferido, se convierte en fijo y no se puede cambiar.

En el siguiente ejemplo se usa la palabra reservada `var` en lugar del tipo de dato explícito, Dart infiere que *a* es `int` al asignarle el valor de 10. Si le asignamos un valor posteriormente distinto a `int` el compilador marcará un error, como en el ejemplo que se le asigna 3.14159.

```
void main() {
  var a = 10;
  a = 15; // OK
  a = 3.14159; // error!
}
```

Tipos de datos

Dart es un lenguaje de programación fuertemente tipado, lo que significa que el tipo de valor que una variable puede contener está definido en el momento de la declaración o inicialización de la variable. En este sentido, Dart ofrece varios tipos de datos fundamentales que son esenciales para la programación. A continuación, se presenta el uso general general de los tipos de datos más utilizados, como `String`, `int`, `double` y `num`.

Tipo de dato String

El tipo *`String`* se utiliza para representar una secuencia de caracteres. Las cadenas en Dart son inmutables, lo que significa que una vez que se crea una cadena, no se puede cambiar. Para usar cadenas de caracteres o Strings:

- Se pueden usar comillas simples o dobles
- Se pueden usar el caracter de escape \
  - 'O\'Connor'
- Se pueden intercalar las comillas sencillas o dobles
  - "O'Connor"
- Se pueden usar triples comillas simples o dobles para String de varias líneas

Interpolación de cadenas

La interpolación de cadenas en un lenguaje de programación es una forma conveniente de insertar o inyectar el valor de una variable, expresión o resultado de una función dentro de una cadena literal (`String`). Esto es especialmente útil para generar cadenas que contienen valores que pueden cambiar durante la ejecución del programa. En Dart, la interpolación de cadenas se realiza utilizando el símbolo de dólar ($) seguido de la variable o expresión que se quiere interpolar. La interpolación se puede dar de la siguiente forma:

- Interpolación simple: dentro de la cadena entrecomillada se le coloca el caracter $ antes del nombre de la variable.

```
void main() {
  String email = ‘micorreo@ucol.mx’;
  print(“Correo: $email”); // Correo: micorreo@ucol.mx
}
```

- Se pueden realizar operaciones, usar funciones o propiedades o métodos dentro de la cadena usando los símbolos `${}`

```
void main() {
  String email = ‘micorreo@ucol.mx’;
  print(“valor: $email Tipo de dato: ${email.runtimeType}”);
  // valor: micorreo@ucol.mx Tipo de dato: String
}
```

Tipado explícito de Strings

En este código de ejemplo se declara una variable email de tipo `String` y se le asigna el valor de manera directa.

```
void main() {
  String email = ‘micorreo@ucol.mx’;
  print(“valor: $email Tipo de dato: ${email.runtimeType}”);
}
```

Tipado inferido de Strings

De igual forma se puede hacer que el tipo `String` se infiera:

```
void main() {
  var email = ‘wmata@ucol.mx’;
  print(“valor: $email Tipo de dato: ${email.runtimeType}”);
}
```

String con múltiples líneas

En el ejemplo de código se muestra cómo en una variable se asigna una página web básica de manera directa, la cual puede ser utilizada durante el programa.

```
void main(List<String> args) {
  String indexHtml = “””
<!DOCTYPE html>
<html lang=”en”>
<head>
    <meta charset=”UTF-8”>
    <meta http-equiv=”X-UA-Compatible” content=”IE=edge”>
    <meta name=”viewport” content=”width=device-width, ini-
```

```
tial-scale=1.0”>
    <title>Document</title>
</head>
<body>

</body>
</html>
“””;

  print(indexHtml);
}
```

Métodos para tipos de datos String

Dart ofrece una rica biblioteca de métodos incorporados para trabajar con cadenas. Estos métodos facilitan la manipulación y consulta de cadenas en diversas formas. A continuación, se presentan en el código una selección de métodos comunes que se utilizan con frecuencia.

```
void main() {
  String email = ‘wmata@ucol.mx’;
  print(email);
  print(‘startsWith wm: ${email.startsWith(‘wm’)}’);
  print(‘endsWith .mx: ${email.endsWith(‘.mx’)}’);
  print(‘contains &: ${email.contains(‘@’)}’);
  print(‘toUpperCase: ${email.toUpperCase()}’);
  print(‘toLowerCase: ${email.toLowerCase()}’);
  print(‘trim: ${‘ wmata@ucol.mx ‘.trim()}’);
  print(‘substring: ${email.substring(6, 10)}’);
}
```

Cada de uno de estos métodos se explican de manera general a continuación:

- startsWith: verifica si una cadena comienza con un cierto subconjunto de caracteres.
- endsWith: verifica si una cadena termina con un cierto conjunto de caracteres.
- contains: se utiliza para verificar si una cadena contiene un cierto subconjunto de caracteres.
- toUpperCase: convierte todos los caracteres de una cadena a mayúsculas.
- toLowerCase: convierte todos los caracteres de una cadena a minúsculas.
- trim: elimina el espacio en blanco desde el inicio y el final de una cadena.
- substring: se utiliza para obtener una subcadena de una cadena existente. Puedes especificar el índice inicial y, opcionalmente, el índice final.

Estos son sólo algunos de los métodos básicos proporcionados por el tipo `String` en Dart. La biblioteca es bastante extensa y ofrece muchas más funcionalidades, como `split`, `replaceAll`, `indexOf`, entre otros, que permiten una manipulación de cadenas más flexible y avanzada.

Propiedades para tipos de datos String

Las cadenas en Dart no sólo vienen con una variedad de métodos útiles para la manipulación, sino que también tienen varias propiedades integradas que nos permiten obtener información sobre la cadena o incluso transformarla de alguna manera. En el siguiente código de ejemplo se describen algunas de las propiedades más comunes.

```
void main() {
  String email = 'wmata@ucol.mx';
  print(email);
  print('length: ${email.length}');
  print('isNotEmpty: ${email.isNotEmpty}');
  print('isEmpty: ${email.isEmpty}');
}
```

Estas propiedades sirven para:

- length: devuelve la longitud de una cadena, es decir, el número de caracteres que contiene.
- isEmpty: devuelve un valor booleano que indica si la cadena está vacía (longitud 0) o no.
- isNotEmpty: es lo opuesto a isEmpty, devuelve true si la cadena no está vacía.

Éstas son sólo algunas de las propiedades básicas proporcionadas por el tipo `String` en Dart. Existen muchas más características y funcionalidades que permiten un manejo más avanzado y flexible de cadenas.

Tipos de datos numéricos

Los tipos de datos numéricos tienen un papel fundamental, ya que se utilizan en casi todos los aspectos de la programación, desde contar y medir hasta realizar cálculos y transformaciones. Dart proporciona principalmente tres tipos de datos numéricos: `int`, `double` y `num`.

Verificación del tipo de dato

La verificación del tipo de dato de una variable se puede realizar utilizando la palabra clave `is`. Esta operación devuelve un valor booleano que indica si la variable es del tipo especificado o no. En el siguiente código se utiliza esta funcionalidad para verificar el tipo de la variable `myNumber`, que se declara de manera explícita como `num`. Recordemos que `num` es una superclase que puede contener tanto valores `int` como `double`.

```
void main() {
  num myNumber = 3.14;
  print(myNumber is double); //true
  print(myNumber is int); //false
  myNumber = 10;
  print(myNumber is double); //false
  print(myNumber is int); //true
}
```

Este código muestra cómo podemos verificar dinámicamente el tipo de una variable en Dart y cómo este tipo puede cambiar si la variable es de un tipo numérico más genérico como num.

Uso de la propiedad runtimeType

Esta propiedad es una característica del lenguaje que permite identificar el tipo de un objeto en tiempo de ejecución. A diferencia de la palabra clave `is`, que se utiliza para verificar si un objeto es una instancia de un tipo particular o de una de sus subclases, `runtimeType` proporciona información precisa sobre el tipo más específico del objeto. Es especialmente útil para la depuración y el registro, así como en situaciones en las que necesitamos realizar acciones basadas en el tipo específico del objeto en tiempo de ejecución.

```
void main() {
  num myNumber = 3.14;
  print(myNumber.runtimeType); //double
  myNumber = 10;
  print(myNumber.runtimeType); //int
}
```

En el código de ejemplo anterior podemos observar cómo nos indica esta propiedad el tipo de dato que adquiere `myNumber` de acuerdo con el valor que tiene asignado.

Tipo de dato int

El tipo `int` se utiliza para representar números enteros. Este tipo de dato es fundamental para contar, iterar y realizar cálculos en los que los valores decimales no son necesarios. Los enteros en Dart son números sin una parte decimal y pueden ser tanto positivos como negativos.

En los siguientes ejemplos declaramos de manera explícita e implícita dos enteros y podemos verificar que el tipo de dato es `int`.

```
void main() {
  int a = 3;
  print(“valor: $a Tipo de dato:${a.runtimeType}”); //int
}

void main() {
  var a = 5;
  print(“valor: $a Tipo de dato:${a.runtimeType}”); //int
}
```

Tipo de dato double

El tipo `double` se utiliza para representar números de punto flotante, es decir, números que tienen una parte decimal. Este tipo de datos es especialmente útil en cálculos que requieren una alta precisión, como cálculos científicos, mediciones y gráficos.

Siguiendo el mismo estilo, ahora en el código presentamos la declaración de dos variables tipo double de manera explícita e implícita respectivamente.

```
void main() {
  double a = 10.5;
  print(“valor: $a Tipo de dato: ${a.runtimeType}”);
}

void main() {
  var a = 10.5;
  print(“valor: $a Tipo de dato: ${a.runtimeType}”);
}
```

Tipo de dato num

Como mencionamos previamente, num es un tipo de dato más generalizado que puede contener tanto int como double. Es precisamente la superclase de estos y es útil cuando quieres que una variable pueda contener cualquiera de los dos tipos.

En los siguientes ejemplos podemos apreciar cómo myNumber y a pueden almacenar cualquier tipo de dato numérico solamente.

```
void main() {
  num myNumber;
  myNumber = 10; // OK
  myNumber = 3.14159; // OK
  myNumber = ‘ten’; // error!
}

void main() {
  num a = 10;
  print(“valor: $a Tipo de dato: ${a.runtimeType}”);
  a = 10.5;
  print(“valor: $a Tipo de dato: ${a.runtimeType}”);
}
```

Métodos y propiedades para tipos de datos numéricos

Los tipos de datos numéricos en Dart (int, double y el tipo más genérico num) ofrecen una serie de métodos y propiedades que permiten realizar diversas operaciones matemáticas y transformaciones. Algunos métodos comunes como floor, ceil, round y truncate se presentan en el siguiente código.

```
void main() {
  int a = 3000;
  double b = 13.5;
```

```
  print(a);
  print(b);
  print('floor: ${b.floor()}');

  print('ceil: ${b.ceil()}');
  print('round: ${b.round()}');
  print('truncate: ${b.truncate()}');
}
```

De manera general, cada uno de estos métodos sirve para:

- floor() redondea el número hacia abajo al entero más cercano.
- ceil() redondea el número hacia arriba al entero más cercano.
- round() redondea el número al entero más cercano, usando el redondeo estándar (redondea hacia arriba si la fracción es igual o mayor a 0.5).
- truncate() elimina la parte fraccional del número, es decir, trunca el número hacia cero.

Operadores aritméticos binarios

Los operadores aritméticos en Dart permiten realizar cálculos matemáticos básicos en las variables numéricas (`int`, `double`, `num`). A continuación presentamos una explicación breve de cada uno de estos.

- Suma (+): este símbolo como operador binario se utiliza para sumar dos operandos.
- Resta (-): es utilizado para restar dos números (el operando derecho menos el operando izquierdo).
- Multiplicación (*): se utiliza para multiplicar los operandos.
- División (/): se utiliza para dividir el operando izquierdo por el operando derecho. La división siempre retorna un tipo `double`, por lo que en ocasiones es necesario realizar un casting o conversión de tipo, posteriormente aprenderemos cómo realizar esta operación.
- Módulo o residuo (%): retorna el residuo de la división del operando izquierdo por el operando derecho.
- División entera (~/): realiza una división, pero retorna el entero inmediato anterior del resultado de la división.

Ejemplo de operaciones básicas con los operadores aritméticos

En el siguiente código se ejemplifica el uso de cada uno de los operadores con los operandos 10 y 3.

```
void main(List<String> args) {
  var n1 = 10;
  var n2 = 3;
```

```
  print(“Suma: $n1 + $n2 = ${n1 + n2}”);
  print(“Resta: $n1 - $n2 = ${n1 - n2}”);
  print(“Multiplicación: $n1 * $n2 = ${n1 * n2}”);
  print(“División: $n1 / $n2 = ${n1 / n2}”);
  print(“Módulo: $n1 % $n2 = ${n1 % n2}”);
  print(“División entera: $n1 ~/ $n2 = ${n1 ~/ n2}”);
}
```

Salida:

```
Suma: 10 + 3 = 13
Resta: 10 - 3 = 7
Multiplicación: 10 * 3 = 30
División: 10 / 3 = 3.3333333333333335
Módulo: 10 % 3 = 1
División entera: 10 ~/ 3 = 3
```

Funciones matemáticas

Dart ofrece la biblioteca estándar `dart:math` que incluye una amplia gama de funciones matemáticas que son útiles para los cálculos numéricos. Para usarla hay que importarla. Algunas de estas funciones se utilizan en el siguiente código de ejemplo:

```
import ‘dart:math’;

void main() {
  print(sin(45 * pi / 180));
  print(cos(135 * pi / 180));

  print(sqrt(5));
  print(pow(5, 2));
  print(max(10, 5));
  print(min(10, 5));
}
```

Salida:

```
0.7071067811865476
-0.7071067811865475
2.23606797749979
25
10
5
```

Operadores unarios

Los operadores unarios en Dart son operadores que actúan sobre un solo operando para realizar diversas operaciones.

- ++ (Incremento): aumenta en 1 el valor de una variable numérica. Para este operador existen dos variantes 1) ++a (pre-incremento) y 2) a++ (post-incremento). La diferencia radica en el momento en que se realiza el incremento en relación con el uso del valor en una expresión.
- −− (Decremento): disminuye en 1 el valor de una variable numérica.

En este código de ejemplo podemos observar el uso de estos operadores.

```
void main() {
  int a = 10;
  a++; // Ahora ‘a’ es 11
  a--; // Ahora ‘a’ es 10
}
```

Operadores de asignación compuesta

Estos operadores combinan una operación matemática con una asignación en una sola expresión, lo que simplifica el código y puede mejorar la legibilidad. Estos operadores realizan una operación específica entre dos operandos y luego asignan el resultado al operando izquierdo. Estos operadores los describimos a continuación:

- Suma y asignación (`+=`): suma el valor del operando de la derecha al de la izquierda y asigna el resultado al operando de la izquierda.
- Resta y asignación (`-=`): resta el valor del operando de la derecha al de la izquierda y asigna el resultado al operando de la izquierda.
- Multiplicación y asignación (`*=`): multiplica el valor del operando de la izquierda por el de la derecha y asigna el resultado al operando de la izquierda.
- División y asignación (`/=`): divide el valor del operando de la izquierda entre el de la derecha y asigna el resultado al operando de la izquierda. El resultado es de tipo double.
- División entera y asignación (`~/=`): realiza una división entera del valor del operando izquierdo por el de la derecha y asigna el resultado al operando de la izquierda.
- Módulo y asignación (`%=`): toma el módulo del operando izquierdo por el operando derecho y asigna el resultado al operando izquierdo.

Ejemplo de incrementos y decrementos con operadores de asignación compuesta

El siguiente código muestra cómo los operadores de asignación compuesta pueden hacer que las operaciones de incremento y decremento sean más concisas. Imprime `1` y `0` en la consola como resultado de las operaciones realizadas.

```
void main() {
  var counter = 0;
  counter += 1; //counter = counter + 1;
  print(counter); //1
  counter -= 1; //counter = counter - 1;
  print(counter); //0
}
```

Ejemplo de incrementos o decrementos con operadores unarios

De igual forma que el ejemplo anterior, el siguiente código ilustra cómo usar operadores unarios para realizar operaciones simples de incremento y decremento, mostrando los resultados `1` y `0` en la consola.

```
void main() {
  var counter = 0;
  counter++;
  print(counter); // 1
  counter--;
  print(counter); // 0
}
```

Ejemplo de multiplicaciones y divisiones con asignación compuesta o cortas

De nueva cuenta, el siguiente código demuestra cómo los operadores de asignación compuesta pueden simplificar las operaciones aritméticas, haciendo el código más conciso. En este caso, los resultados `30.0` y `15.0` son impresos en la consola.

```
void main() {
  double myValue = 10;
  myValue *= 3; // myValue = myValue * 3;
  print(myValue); // 30.0;
  myValue /= 2; // myValue = myValue / 2;

  print(myValue); // 15.0;
}
```

Casting y operaciones con distintos tipos de datos

El casting se refiere al proceso de convertir un valor de un tipo de dato a otro. Dart proporciona métodos específicos para realizar estas conversiones entre tipos de datos numéricos y cadenas de texto (strings).

En el siguiente código de ejemplo se hace uso de algunos métodos de las clases numéricas que permiten la conversión de tipo.

```
void main() {
  int a = 10;
  double b = 3.3;
  print(a.toDouble());
  print(b.toInt());
  print(“Número: “ + a.toString());
}
```

Los métodos que usamos fueron:

- `toDouble()` convierte el valor de la variable a de tipo `int` a un valor de tipo `double`.
- `toInt()` convierte el valor de la variable b de tipo `double` a un valor de tipo `int`.
- `toString()` convierte el valor de la variable a de tipo `int` a `String`, lo cual permite la concatenación o unión con la cadena “Número: “, para formar una sola cadena, que se muestra en la consola como “Número: 10”.

En el siguiente código de ejemplo mostramos cómo convertir cadenas de texto (strings) a tipos numéricos y cómo formatear números decimales a una precisión específica. La conversión de cadenas a números se realiza utilizando métodos específicos como `int.parse()` y `double.parse()`.

```
void main() {
  String a = “10”;
  String b = “3.3”;
  print(int.parse(a) * 2);
  print(double.parse(b) * 2);
  double pi = 3.1416;
  print(pi.toStringAsFixed(2));
}
```

De acuerdo con el código anterior, tenemos que los métodos:

- `int.parse(a)` convierte la cadena *“10”* en un entero *10*. Luego, este número se multiplica por *2*, y el resultado *20* se imprime en la consola.
- `double.parse(b)` convierte la cadena *“3.3”* en un número de punto flotante *3.3*. Este número se multiplica por *2*, y el resultado *6.6* se imprime en la consola.
- `toStringAsFixed(2)` se usa para formatear el número de punto flotante en una cadena con 2 decimales. En este caso, *3.14* es el valor que se imprime en la consola.

Biblioteca de internacionalización de Dart

El paquete `intl` en Dart es una biblioteca que proporciona internacionalización y localización para las aplicaciones. Este paquete es especialmente útil cuando necesitamos adaptar la aplicación para múltiples idiomas y regiones, proporcionando funcionalidades como el formateo de fechas, números y cadenas, además de soporte para pluralización y mensajes localizados.

Para instalar el paquete `intl` desde la consola o terminal escribimos lo siguiente:

```
C:\>dart pub add intl
Resolving dependencies...
+ clock 1.1.1
+ intl 0.17.0
Changed 2 dependencies!
```

En el siguiente ejemplo de código formateamos un número decimal mediante la clase `NumberFormat` para personalizar la representación de números decimales en un formato específico.

```
import 'package:intl/intl.dart';

void main() {
  var mascara = NumberFormat('#,###,000.00');
  print(mascara.format(2563.2562));
}
```

- En la primera línea importamos el paquete, el cual contiene a la clase `NumberFormat`.
- En la línea donde se declara la variable mascara, es donde se define la máscara de formateo `#,###,000.00` para los números. Esta máscara tiene varias partes:
  - `#,###`: indica que se deben usar comas como separadores de miles.
  - `000`: asegura que siempre haya al menos tres dígitos antes del punto decimal.
  - `.00`: especifica que se deben mostrar exactamente dos decimales.
- Finalmente, el número 2563.2562 se formatea utilizando la máscara definida y se imprime en la consola. Dado que la máscara especifica que se deben mostrar exactamente dos decimales, el número se redondea a 2563.26. Además, la máscara también asegura que haya al menos tres dígitos antes del punto decimal, por lo que no se añaden ceros adicionales en este caso. El resultado será 2,563.26.

Asignación de distintos tipos de datos numéricos

En el siguiente ejemplo de código intentamos asignar un valor decimal a una variable de tipo entero, lo cual resulta en un error de compilación. En Dart, no se puede asignar directamente un valor de tipo `double` a una variable de tipo `int` sin realizar una conversión explícita. Dart es un lenguaje de tipado fuerte y no realiza la conversión automática entre tipos de datos numéricos incompatibles.

```
void main() {
  var integer = 100;
  var decimal = 12.5;
  integer = decimal; //error!
}
```

Tipo de dato de variables num

En el siguiente código, mostramos nuevamente el uso de la propiedad `runtimeType` para verificar el tipo de datos en tiempo de ejecución. Declaramos dos variables `integer` y `decimal` de tipo `num`, que como hemos dicho es un tipo numérico general que puede contener tanto enteros como decimales. Al final, podremos ver que el `runtimeType` de `integer` cambia a `double` después de la asignación, mientras que el de `decimal` permanece como `double`.

```
void main() {
  num integer = 100;
  print(integer.runtimeType);
  num decimal = 12.5;
  print(decimal.runtimeType);
  integer = decimal;
  print(integer.runtimeType);
  print(decimal.runtimeType);
}
```

Ahora en el siguiente código creamos nuevamente las dos variables integer y decimal, ambas de tipo num. Después de la asignación, veremos que el runtimeType de *decimal* cambia a int, mientras que el de integer permanece como int, demostrando que en Dart, el tipo de datos en tiempo de ejecución de una variable tipo num puede cambiar dependiendo del tipo de valor numérico que contenga.

```
void main() {
  num integer = 100;
  print(integer.runtimeType);
  num decimal = 12.5;
  print(decimal.runtimeType);
  decimal = integer;
  print(integer.runtimeType);
  print(decimal.runtimeType);
}
```

Casting int a double

En el siguiente ejemplo mostramos cómo convertir un número de tipo `int` a `double` utilizando el método `toDouble()`. Al ejecutar el código, veremos que `numeroEntero` sigue siendo *3*, pero `numeroDouble` será *3.0*. Numéricamente hablando ambas variables son iguales, pero de diferente tipo de dato, es decir, `numeroEntero` es de tipo `int`, y `numeroDouble` es de tipo `double`.

```
void main() {
  num numeroEntero = 3;
```

```
  num numeroDouble = numeroEntero.toDouble();
  print(numeroEntero);
  print(numeroDouble);
}
```

Tipo de dato lógico (Booleano)

El tipo de dato lógico o booleano es un tipo de dato fundamental en Dart y en la mayoría de los lenguajes de programación. En Dart, se utiliza la palabra reservada `bool` para declarar variables booleanas. Estas variables pueden tener sólo dos posibles valores: `true` (verdadero) o `false` (falso). Los booleanos son útiles para controlar el flujo del programa mediante estructuras condicionales como `if`, `while`, y `for`. La negación de una variable booleana se realiza utilizando el carácter de exclamación ! justo antes de la variable o la expresión booleana. Esto invierte el valor de la variable o la expresión; es decir, true se convierte en false y viceversa. A continuación presentamos un breve esquema que resume a este tipo de dato.

- Se usa la palabra reservada bool
- Tiene dos posibles valores
  - true
  - false
- Para negación se usa el caracter ! antes de la variable

Tipado explícito de variables bool

En el siguiente código mostramos cómo declarar de manera explícita una variable lógica.

```
void main(List<String> args) {
  bool esNegativo = true;

  print(esNegativo); //true
}
```

Tipado inferido de variables bool

Ahora en el siguiente código mostramos cómo declarar de manera implícita (inferencia) una variable lógica. Como podemos observar, al igual que con los otros tipos de datos, al usar la palabra reservada `var`, es obligatorio asignar un valor para fijar el tipo de dato correspondiente, que en este caso es `bool`.

```
void main(List<String> args) {
  var esNegativo = true;

  print(esNegativo);
}
```

null safety

Se dice que Dart es null safety, ésta es una característica de Dart que tiene como objetivo eliminar o minimizar la presencia de errores relacionados con valores null en tiempo de ejecución. Antes de la introducción de esta característica al lenguaje, el acceso a una propiedad o la llamada a un método en un objeto null resultaba en un error en tiempo de ejecución, lo que provocaba comportamientos inesperados o fallas en la aplicación.

Con null Safety, el lenguaje se diseña de tal forma que las variables no pueden contener null a menos que se declare explícitamente que pueden hacerlo. Esto se logra mediante la sintaxis de tipo nullable, en la que se añade un signo de interrogación (?) después del tipo de dato durante la declaración de la variable. Por ejemplo, String? indica que una variable puede contener una cadena o null.

En el siguiente código de ejemplo hacemos uso de esta característica haciendo una variable booleana nullable llamada esNegativo. La variable se declara con un signo de interrogación (bool?), lo que significa que puede contener tres valores posibles: true, false o null. Como la variable esNegativo no se inicializa con un valor, su valor predeterminado será null, valor que se imprime en la consola al ejecutar el programa.

```
void main(List<String> args) {
  bool? esNegativo;

  print(esNegativo);
}
```

Operadores de comparación

Estos operadores se utilizan para comparar dos valores y determinar la relación entre ellos. Devuelven un valor booleano (true o false) en función del resultado de la comparación. Los operadores de comparación comunes son:

- Igual (==): verifica si dos valores son iguales.
- No Igual o diferente (!=): verifica si dos valores son diferentes.
- Menor que (<): verifica si el valor del lado izquierdo es menor que el valor del lado derecho.
- Menor o Igual (<=): verifica si el valor del lado izquierdo es menor o igual al valor del lado derecho.
- Mayor que (>): verifica si el valor del lado izquierdo es mayor que el valor del lado derecho.
- Mayor o Igual (>=): verifica si el valor del lado izquierdo es mayor o igual al valor del lado derecho.

Operadores lógicos

Los operadores lógicos en Dart permiten combinar múltiples expresiones booleanas y retornar un único valor booleano. Son importantes para la construcción de estructuras de control como condicionales y ciclos o bucles. Los operadores lógicos más comunes en Dart son && (AND), || (OR) y como habíamos mencionado previamente ! (NOT).

Operador AND (&&)

El operador && retorna `true` si ambas expresiones son verdaderas. Si alguna de las expresiones es `false`, el resultado será `false`.

A continuación, presentamos un código que genera la tabla de verdad de && (AND).

```
void main(List<String> args) {
  print(“a \t b \t a and b”);
  print(“false \t false \t ${false && false}”);
  print(“false \t true \t ${false && true}”);
  print(“true \t false \t ${true && false}”);
  print(“true \t true \t ${true && true}”);
}
```

Salida:

```
a        b       a and b
false    false   false
false    true    false
true     false   false
true     true    true
```

Operador OR (||)

El operador || retorna `true` si al menos una de las expresiones es verdadera. Si ambas expresiones son `false`, el resultado será `false`.

A continuación, presentamos un código que genera la tabla de verdad de || (OR).

```
void main(List<String> args) {
  print(“a \t b \t a or b”);
  print(“false \t false \t ${false || false}”);
  print(“false \t true \t ${false || true}”);
  print(“true \t false \t ${true || false}”);
  print(“true \t true \t ${true || true}”);
}
```

Salida:

```
a        b       a or b
false    false   false
false    true    true
true     false   true
true     true    true
```

Operador NOT (!)

El operador ! invierte el valor de una expresión booleana. Si la expresión es `true`, el operador ! la convierte en `false` y viceversa.

A continuación presentamos un código que genera la tabla de verdad de `! (NOT)`.

```
void main(List<String> args) {
  print(“a \t not a”);
  print(“false \t ${!false}”);
  print(“true \t ${!true}”);
}
```

Salida:

```
a        not a
false    true
true     false
```

Ejemplo usando operadores lógicos

El siguiente código de ejemplo genera 10 números aleatorios entre 0 y 10 utilizando la clase `Random` de la biblioteca `dart:math`. Con un ciclo `for` (el cual estudiaremos más a detalle posteriormente), evalúa cada número generado. Si el número está en el rango [6,10], imprime que la calificación es *aprobatoria*. Si el número está en el rango de [0, 5], imprime que la calificación es *reprobatoria*.

```
import ‘dart:math’;

void main(List<String> args) {
  Random numRandom = Random();
  numRandom.nextInt(11);
  for (var i = 0; i < 10; i++) {
    var cal = numRandom.nextInt(11);

    if (cal > 5 && cal < 11) {
      print(“$cal es aprobatoria”);
    } else {
      print(“$cal es reprobatoria”);
    }
  }
}
```

Salida:

```
4 es reprobatoria
8 es aprobatoria
7 es aprobatoria
0 es reprobatoria
2 es reprobatoria
7 es aprobatoria
```

```
1 es reprobatoria
7 es aprobatoria
10 es aprobatoria
7 es aprobatoria
```

Tipado de datos dinámico (dynamic)

En Dart, el tipo de dato `dynamic` es un tipo especial que indica que una variable puede contener cualquier tipo de dato. Es decir, una variable declarada como tal, puede ser asignada y reasignada con valores de diferentes tipos en tiempo de ejecución sin generar algún error de tipo de dato. Es importante considerar que el uso de `dynamic` trae consigo riesgos y debería emplearse con precaución, pues al permitir cualquier tipo de valor, se pierden las ventajas del sistema de tipado estático de Dart, como la detección temprana de errores y las optimizaciones de rendimiento. Por lo tanto, generalmente es recomendable especificar el tipo de datos siempre que sea posible y recurrir a `dynamic` sólo cuando sea estrictamente necesario. No usar `dynamic` o tipos de datos generales como `num` siempre ayudará a evitar efectos colaterales.

En el siguiente programa declaramos múltiples variables de diferentes tipos (int, double, num, String, bool, y dynamic) y las inicializamos con valores. Luego crea una lista misVariables que contiene todas estas variables. La función forEach() se utiliza para iterar a través de la lista misVariables. Dentro de la función anónima (e) => ..., se imprime el valor de cada elemento junto con su tipo de dato utilizando la propiedad runtimeType, mostrando la información de cada variable en la consola. Las instrucciones o conceptos que no hemos visto hasta el momento, posteriormente los estudiaremos.

```
void main(List<String> args) {
  var esNegativo;

  print(esNegativo);
}

void main() {
  int varInt = 1;
  double varDouble = 1.1;
  num varNum = 1.0;
  String varString = “Hola”;
  bool varBool = true;
  dynamic varDynamic = “Mundo”;
  var misVariables = [
    varInt,
    varDouble,
    varNum,
    varString,
```

```
    varBool,
    varDynamic
  ];
  misVariables.forEach((e) => print("valor: $e Tipo de dato:${e.
runtimeType}"));
}
```

Salida:

```
valor: 1 Tipo de dato:int
valor: 1.1 Tipo de dato:double
valor: 1.0 Tipo de dato:double
valor: Hola Tipo de dato:String
valor: true Tipo de dato:bool
valor: Mundo Tipo de dato:String
```

Uso del tipo de dato null (null safety)

El siguiente código tiene un problema relacionado con la inicialización de variables. En Dart, las variables de tipo *no* `nullable` (como `int`, `double`, `String`, etc.) deben ser inicializadas antes de su uso, o de lo contrario se producirá un error en tiempo de ejecución. En este caso, la línea `int a;` se declara `a` pero no se inicializa. Cuando intentamos imprimir con `print(a);` Dart lanza un error porque estás intentando usar una variable no inicializada.

```
void main(List<String> args) {
  int a;

  print(a); //error
}
```

Hay varias formas de solucionar este problema:

- Inicializar la variable al declararla:

```
int a = 0;
```

- Utilizar el tipo `nullable` (`int?`) si se va a permitir eventualmente pueda ser `null`:

```
int? a;
```

- Inicializar la variable en cualquier lugar antes de usarla:

```
int a;
a = 0;
```

- Verificar si la variable es `null` antes de utilizarla:

```
int a = 0;
  if (a != null){
  print(a);
}
```

Palabras reservadas Final y Const

En Dart, las palabras clave `final` y `const` se utilizan para declarar variables que no se pueden modificar una vez que se les ha asignado un valor. Aunque ambos tienen un comportamiento similar, existen diferencias sutiles entre ellas.

Primero veamos `final`:

- Cuando declaramos una variable como `final`, ésta no puede cambiar su valor una vez que inicializada, se le llaman variables inmutables.
- No es necesario que un valor final se conozca en tiempo de compilación; puede determinarse en tiempo de ejecución.
- Es ideal para variables cuyo valor debe calcularse, pero no debe cambiar posteriormente. Su valor se puede generar mediante expresiones.
- Sirven para declarar variables que no van a cambiar durante la ejecución del programa.

Ahora `const`:

- Se utiliza para valores que son constantes en todo el programa, no se puede modificar su valor.
- Mientras que `final` se puede utilizar para miembros de instancia (en el contexto de la programación orientada a objetos), lo que significa que el valor puede ser diferente para cada instancia, pero aun así no se podrá cambiar una vez asignado.
- Se crean en tiempo de compilación.

En el siguiente código se declaran dos variables enteras y se les asignan su valor. A la variable `final nFinal` se le asigna un valor constante en tiempo de compilación que depende de los valores de `n1` y `n2`, que en este caso es 6. Si posteriormente tuviéramos la necesidad de cambiar su valor, sería imposible. Intenta cambiar el valor y verás el error que te marca.

```
void main() {
  int n1 = 2;
  int n2 = 3;
  final nFinal = n1 * n2;
  print(nFinal); //6
}
```

Si quisiéramos hacer lo mismo con `const`, podemos ver en el siguiente código que es imposible asignarle un valor a la variable `nConst` en tiempo de compilación con las mismas variables.

```
void main() {
  int n1 = 2;
  int n2 = 3;
  const nConst = n1 * n2; //error
  print(nConst);
}
```

En este caso es obligatorio inicializarlo con un valor constante, por ejemplo 10. De nueva cuenta no se puede cambiar el valor de la variable, es inmutable.

```
void main() {
  const nConst = 10;
  print(nConst);
}
```

Es posible darle valor a una variable `final` en tiempo de ejecución. En el siguiente programa solicitamos al usuario su nombre y edad a través de la entrada estándar (teclado) y mostramos los mensajes en la salida estándar (consola o terminal).

```
import 'dart:io';

void main() {
  print("Dame tu nombre: ");
  final nombre = stdin.readLineSync();
  print("Tu nombre es $nombre\n");
  //nombre = "Juan";
  stdout.write("Dame tu edad:");
  final edad = stdin.readLineSync();
  stdout.write("Hola $nombre, tienes $edad años\n");
}
```

En este programa tenemos algunas instrucciones nuevas que explicaremos a continuación:

- `import 'dart:io';` sirve para importar la biblioteca para usar funcionalidades de E/S (entrada/salida) como `stdin` y `stdout`.
- `print("Dame tu nombre: ");` imprime el mensaje en la consola pidiendo el nombre del usuario.
- `final nombre = stdin.readLineSync();` lee una línea desde el teclado (`stdin`) y la guarda en la variable `nombre`. La palabra clave `final` indica, como mencionamos anteriormente, que esta variable no puede ser modificada posteriormente en el programa.
- `print("Tu nombre es $nombre\n");` imprime el nombre recibido, utilizando interpolación de cadenas.
- La línea `//nombre = "Juan";` está comentada para que confirmes que, si se vuelve a asignar un valor a la variable, producirá un error de compilación porque es una variable `final` y, por lo tanto, no puede ser modificada.
- El resto del programa sigue los mismos pasos para preguntar la edad y desplegar un mensaje final indicando el nombre y la edad que fueron capturados.

Capítulo 3
Estructuras condicionales

En este capítulo revisamos las estructuras condicionales, fundamentales para controlar el flujo de ejecución de un programa y tomar decisiones basadas en ciertas condiciones. Exploraremos la sentencia `if` y sus diferentes variantes, que nos permiten ejecutar bloques de código en función de si una condición es verdadera o falsa. Analizaremos ejemplos prácticos de su aplicación, desde situaciones simples hasta anidaciones complejas.

Además, exploraremos el operador ternario, una forma concisa de expresar estructuras condicionales en una sola línea. Aprenderemos cómo determinar resultados en función de diferentes condiciones y cómo asignar estos resultados a variables de manera eficiente. También exploraremos la sentencia switch-case, una herramienta poderosa para tomar decisiones basadas en múltiples casos.

A lo largo de esta sección, veremos ejemplos detallados que ilustran cómo aplicar estas estructuras condicionales en situaciones prácticas, proporcionando una comprensión sólida y práctica de cómo controlar el flujo de un programa en Dart.

Sentencia if

El bloque if-else en Dart es una estructura de control de flujo que se utiliza para ejecutar un bloque de código si una condición dada es verdadera (`true`), y otro bloque de código si la condición es falsa (`false`). Aquí está la sintaxis básica y detallada:

```
if (condición) {

}else{

}
```

Ejemplo if simple

En el siguiente programa verificamos si la calificación almacenada en la variable entera cal es aprobatoria (mayor o igual que 6) y, en caso afirmativo, imprime el mensaje Calificación aprobatoria en la consola.

```
void main() {
  int cal = 8;
  if (cal >= 6) {
    print(“Calificación aprobatoria”);
  }
}
```

Salida:

```
D:\mi_app>dart run
Building package executable...
Built mi_app:mi_app.
Calificación aprobatoria
```

Ejemplo if-else

En el siguiente programa utilizamos la misma variable entera `cal` con un valor de 5 y una sentencia `if-else` para evaluar si la calificación es aprobatoria. Como 5 no es mayor que 6, se imprime `Calificación reprobatoria` en la consola.

```
void main() {
  int cal = 5;
  if (cal > 6) {
    print(“Calificación aprobatoria”);
  } else {
    print(“Calificación reprobatoria”);
  }
}
```

Salida:

```
Calificación reprobatoria
```

Ejemplos if anidados

Las condiciones también podemos anidarlas, en el siguiente programa utilizamos la estructura de control `if-else` anidada para comparar los números contenidos en las variables enteras num1 y num2. Esta estructura condicional tiene tres posibles caminos o ramas:

- Primero `if (num1 > num2)` evalúa si num1 es mayor que num2. Si es verdadero, imprime `El mayor es num1`.
- Si la primera condición es falsa, pasa a la condición `else if (num1 < num2)`, que evalúa si num1 es menor que num2. Si es verdadero, imprime `El mayor es num2`.
- Por último, si ninguna de las condiciones anteriores es verdadera, se ejecuta el `else`, que imprime `num1 y num2 son iguales`.

En este caso, num1 y num2 son iguales (ambos son 5), por lo que se imprime `num1 y num2 son iguales`.

```
void main() {
  int num1 = 5;
  var num2 = 5.0;
  if (num1 > num2) {
    print(“El mayor es num1”);
  } else if (num1 < num2) {
```

```
    print(“El mayor es num2”);
  } else {
    print(“num1 y num2 son iguales”);
  }
}
```

Salida:

```
num1 y num2 son iguales
```

En el siguiente programa anidamos más estructuras de control `if-else-if` para evaluar el valor de la variable n. Dependiendo del valor de ésta, imprime el número en forma de palabra (uno, dos, `tres`, `cuatro` o `cinco`). En este caso específico, como `n = 5`, el programa imprimirá `cinco`.

```
void main() {
  var n = 5;
  if (n == 1) {
    print(“uno”);
  } else if (n == 2) {
    print(“dos”);
  } else if (n == 3) {
    print(“tres”);
  } else if (n == 4) {
    print(“cuatro”);
  } else {
    print(“cinco”);
  }
}
```

Ejemplo para obtener el mayor de dos números con interpolación de cadenas

En el siguiente programa utilizamos la estructura de control o condicional `if-else` para comparar dos números enteros almacenados en las variables `n1` y `n2`. Usamos la interpolación de cadenas para inyectar los valores de estas variables directamente en la cadena que se imprime. En este caso específico, dado que `n1 = 3` y `n2 = 5`, la salida será `5 > 3`.

```
void main() {
  int n1 = 3, n2 = 5;
  if (n1 > n2) {
    print(“$n1 > $n2”);
  } else {
    print(“$n2 > $n1”);
  }
}
```

Operador ternario

El operador ternario en Dart es una forma concisa de realizar una operación condicional. La sintaxis es `condición ? expresión1 : expresión2`. Si la `condición` es verdadera, se evalúa y devuelve `expresión1`; de lo contrario, se evalúa y devuelve `expresión2`. Es una forma más corta y legible de escribir una estructura `if-else` simple.

Retomando el ejemplo para obtener el mayor de dos números, podemos ver que el código se extiende por la estructura condicional. Al ser `9 > 1`, la salida del programa es `El mayor es: 9`.

```
void main() {
  int n1 = 9, n2 = 4;
  int mayor;
  if (n1 > n2) {
    mayor = n1;
  } else {
    mayor = n2;
  }
  print(“El mayor es: $mayor”);
}
```

Salida:

```
El mayor es: 9
```

Haciendo uso del operador ternario, podemos ver el código es más sencillo y comprensible, como los datos son los mismos que en el ejemplo anterior, el resultado es el mismo.

```
void main() {
  int n1 = 9, n2 = 4;
  int mayor;

  n1 > n2 ? mayor = n1 : mayor = n2;

  print(“El mayor es: $mayor”);
}
```

Salida:

```
El mayor es: 9
```

Podemos también asignar el resultado de la condición a una variable, en este caso se asigna a la variable entera mayor.

```
void main() {
  int n1 = 9, n2 = 4;

  int mayor = n1 > n2 ? n1 : n2;
```

```
  print(“El mayor es: $mayor”);
}
```

Salida:

```
El mayor es: 9
```

Podemos usar la expresión interpolada en una cadena, en el siguiente ejemplo el resultado se genera directamente dentro de la cadena.

```
void main() {
  int n1 = 9, n2 = 4;

  print(“El mayor es: ${n1 > n2 ? n1 : n2}”);
}
```

Salida:

```
El mayor es: 9
```

El siguiente código ejecuta una comparación entre dos números enteros, `n1` y `n2`, determinando cuál de los dos es el menor. Se presenta esta lógica de comparación de cuatro formas diferentes, todas con el mismo objetivo:

1. Utilizar una estructura de control `if-else` tradicional.
2. Aplicar el operador ternario de manera explícita para asignar el valor a la variable `menor`.
3. Emplear el operador ternario de manera compacta para realizar la asignación directamente en la declaración de la variable `menor`.
4. Imprimir el resultado directamente dentro de una plantilla de cadena inyectando el operador ternario. Esta línea demuestra cómo el operador ternario puede ser utilizado dentro de la interpolación de cadenas para imprimir el resultado de manera directa y concisa.

```
void main() {
  int n1 = 10;
  int n2 = 5;
  int menor;
  if (n1 < n2) { //forma 1
    menor = n1;
  } else {
    menor = n2;
  }
  print(“El menor es: $menor”);

  n1 < n2 ? menor = n1 : menor = n2; //forma 2
  print(“El menor es: $menor”);
  menor = n1 < n2 ? n1 : n2; //forma 3
  print(“El menor es: $menor”);
  print(“El menor es: ${n1 < n2 ? n1 : n2}”); // forma 4
}
```

Salida:

```
El menor es: 5
El menor es: 5
El menor es: 5
El menor es: 5
```

Sentencia switch-case

La sentencia `switch-case` es una estructura de control que permite ejecutar diferentes bloques de código en función del valor de una variable dada. Es similar a una serie de `if-else-if` pero más legible cuando se tienen múltiples condiciones que comparar contra una sola variable. Su sintaxis general es:

```
switch (expresión) {
    case valor1:
      break;
    case valor2:
      break;
      . . .
    case valorn:
      break;
    default:
}
```

Donde:

- Se inicia con la palabra reservada `switch` seguida de la variable que se quiere comparar.
- Después del switch, se definen las sentencias `case` necesarias que representen los posibles valores que puede tener la variable. Si la variable concuerda o hace *match* con uno de estos valores, se ejecuta el bloque de código asociado a ese `case`.
- Cada sentencia `case` debe terminar con la palabra reservada `break`, para evitar que el programa continúe ejecutando los casos siguientes. El break hace que el flujo del programa salga de la estructura `switch`.
- Por último, la sentencia `default` es opcional y se ejecuta siempre y cuando ninguna de las sentencias `case` coincida con el valor de la variable. Funciona como una especie de `else` final en una serie de `if-else-if`.

En el siguiente programa podemos ver que la sentencia `switch-case` asocia un número entero con un día de la semana. Al evaluar la variable `dia` con el valor `7`, la estructura encuentra la coincidencia (hace *match*) con el caso correspondiente al `Domingo` y lo imprime. Si no hubiera coincidencia, se ejecutaría la opción por defecto en `default`, imprimiendo `Día desconocido`.

```
void main() {
  int dia = 7;

  switch (dia) {
    case 1:
      print(“Lunes”);
      break;
    case 2:
      print(“Martes”);
      break;
    case 3:
      print(“Miércoles”);
      break;
    case 4:
      print(“Jueves”);
      break;
    case 5:
      print(“Viernes”);
      break;
    case 6:
      print(“Sábado”);
      break;
    case 7:
      print(“Domingo”);
      break;
    default:
      print(“Día desconocido”);
      break;
}
```

Salida:

```
Domingo
```

Utilizando la variable ahora como *String*, podemos mapear los nombres de los días de la semana a sus correspondientes números ordinales en la semana. Ahora la variable `dia` se establece como `sábado`, la cual coincide con el caso de `sábado`, y se ejecuta el código que imprime el número 6. De igual forma que en el ejemplo anterior, si `dia` no coincidiera con ninguno de los casos definidos, se ejecutaría el caso por defecto, imprimiendo Día desconocido.

```
void main() {
  var dia;
  dia = “sábado”;

  switch (dia) {
    case “lunes”:
      print(1);
      break;
    case “martes”:
      print(2);
      break;
    case “miércoles”:
      print(3);
      break;
    case “jueves”:
      print(4);
      break;
    case “viernes”:
      print(5);
      break;
    case “sábado”:
      print(6);
      break;
    case “domingo”:
      print(7);
      break;
    default:
      print(“Día desconocido”);
  }
}
```

Salida:

```
6
```

¿Qué pasa si a la viariable `dia` se le asigna una cadena sin acento, o con mayúscula inicial? En el siguiente programa se asignan números a los días de la semana. Si la variable `dia` es igual a alguna de las variaciones de `miércoles` (con diferente capitalización o acentuación), el programa imprimirá el número 3. Las otras condiciones verifican diferentes días y, si hay coincidencia, imprimen el número correspondiente a cada día de la semana. Si el día no es reconocido, el programa imprime “Día desconocido”. Podemos observar que `break` termina la opción y sale del `switch`, se puede aprovechar esto para permitir varios casos similares.

```
void main() {
cvar dia;
  dia = “Miércoles”;

  switch (dia) {
    case “lunes”:
      print(1);
      break;
    case “martes”:
      print(2);
      break;
    case “miércoles”:
    case “miercoles”:
    case “Miércoles”:
    case “Miercoles”:
      print(3);
      break;
    case “jueves”:
      print(4);
      break;
    case “viernes”:
      print(5);
      break;
    case “sábado”:
      print(6);
      break;
    case “domingo”:
      print(7);
      break;
    default:
      print(“Día desconocido”);
  }
}
```

Salida:

```
3
```

En el siguiente programa realizamos la misma tarea que en el anterior, asignando un número a cada día de la semana. La diferencia principal es que transforma el contenido de la variable `dia` a mayúsculas antes de evaluarla en la sentencia `switch`, mediante el método `toUpperCase()`. Esto nos ayuda a simplificar las comparaciones dentro de los casos al no tener que verificar todas las combinaciones de mayúsculas y minúsculas o acentuación para

miércoles y sábado. Ahora sólo se necesita verificar la versión en mayúscula de cada día, con y sin acentos. Esto hace que el código sea un poco más eficiente y fácil de leer y mantener.

```
void main() {
  var dia;
  dia = “sábado”;

  switch (dia.toString().toUpperCase()) {
    case “LUNES”:
      print(1);
      break;
    case “MARTES”:
      print(2);
      break;
    case “MIERCOLES”:
    case “MIÉRCOLES”:
      print(3);
      break;
    case “JUEVES”:
      print(4);
      break;
    case “VIERNES”:
      print(5);
      break;
    case “SABADO”:
    case “SÁBADO”:
      print(6);
      break;
    case “DOMINGO”:
      print(7);
      break;
    default:
      print(“Día desconocido”);
  }
}
```

Salida:

```
6
```

Ahora vamos a introducir el uso de una enumeración (enum), la cual es una forma de definir un tipo de dato con un conjunto fijo y limitado de constantes, que en este caso son los días de la semana. Declaramos un enum llamado Dia con los días como miem-

bros. Este enfoque con enum aumenta la claridad del código y evita errores comunes en comparaciones de cadenas, tales como errores de tipografía o problemas de mayúsculas y minúsculas, ya que se trabaja con un tipo específico de dato y no con cadenas de texto.

```
enum Dia {
  lunes,
  martes,
  miercoles,
  jueves,
  viernes,
  sabado,
  domingo,
}

void main() {
  var dia;
  dia = 0;
  print(dia);
  print(Dia.lunes.index);

  dia = Dia.martes;
  switch (dia) {
    case Dia.lunes:
      print(1);
      break;
    case Dia.martes:
      print(2);
      break;
    case Dia.miercoles:
      print(3);
      break;
    case Dia.jueves:
      print(4);
      break;
    case Dia.viernes:
      print(5);
      break;
    case Dia.sabado:
      print(6);
      break;
    case Dia.domingo:
      print(7);
      break;
    default:
      print(“Día desconocido”);
  }
}
```

Salida:

```
0
0
2
```

Ámbito o scope

El ámbito o *scope* en programación se refiere al contexto en el que una variable o función es accesible dentro de un programa. Las variables y funciones sólo pueden ser accedidas desde dentro del contexto en el que fueron definidas. Este concepto es fundamental para la gestión de la memoria y la legibilidad del código, así como para evitar colisiones de nombres en diferentes partes de un programa. Analicemos el siguiente código de ejemplo:

- En el código proporcionado, el ámbito de la variable `numero` y la variable `status` se define dentro de la función `main`, lo que significa que ambas variables pueden ser accedidas en cualquier parte de esta función. La variable `numero` se inicializa con el valor 500.
- Dentro de la estructura condicional `if`, se verifica si `numero` es igual a `500`. Si es así, la variable `status` se establece como `true`, y se imprime `ok` seguido del valor de `status`. En caso contrario, se crea una nueva variable `status` con el mismo nombre en el ámbito del bloque `else`. Esta nueva variable es una instancia diferente y sólo existe dentro de este bloque. Se le asigna `false` y se imprime `fail` seguido del valor de esta nueva variable `status`.
- Después del bloque condicional, se imprime la variable `status` definida como mencionamos previamente, en el ámbito de la función `main`. Debemos hacer notar que la variable `status` dentro del bloque `else` no afecta a la variable `status` del ámbito de `main` debido a que son variables diferentes, que existen en distintos ámbitos.
- Por último, en la condición dejamos el comentario `501`. Prueba cambiando el valor a este en la condición y notarás que el resultado es distinto, pues el programa imprime ahora `fail` en el bloque `else`, pero la última línea `print(status);` imprime `null` debido a que `status` no fue inicializado en el bloque `if`, el cual establece `status` a `true`.

```
void main(List<String> args) {
  var numero = 500;
  bool? status;

 if (numero == 500 /*501*/) {
    status = true;
    print(“ok $status”);
  } else {
    bool status = false;
    print(“fail $status”);
  }
  print(status);
}
```

Salida:

```
ok true
true
```

Capítulo 4
Estructuras repetitivas (ciclos o loops)

En este capítulo utilizaremos las estructuras repetitivas, también conocidas como ciclos o loops. Estas estructuras son fundamentales para automatizar tareas repetitivas y ejecutar bloques de código de manera eficiente y controlada. Exploraremos diferentes tipos de ciclos: 1) el ciclo `for` que nos permite especificar un número de iteraciones, 2) el ciclo `while` que se ejecuta mientras una condición sea verdadera, y 3) el ciclo `do-while` que garantiza al menos una ejecución. Exploraremos además técnicas avanzadas como ciclos anidados, generadores y reductores. Continuaremos a lo largo de este capítulo con ejemplos detallados que te ilustrarán cómo aplicar estas estructuras repetitivas en situaciones prácticas, proporcionando una comprensión sólida y práctica de cómo automatizar procesos en tus aplicaciones.

Estructura repetitiva for

El ciclo `for` en programación se utiliza para ejecutar un bloque de código repetidamente un número determinado de veces. La sintaxis de este ciclo generalmente incluye tres partes:

1. Inicialización: Establecemos un contador y lo inicializamos con un valor, por ejemplo `for(int i = 0; ...)` inicia un contador `i` en `0`.
2. Condición: Definimos una condición que se va a evaluar antes de cada iteración del ciclo. Si la condición es verdadera, el bloque de código dentro del ciclo se ejecuta. Si es falsa, el ciclo termina, por ejemplo `for(...; i < 10; ...)` se va a repetir hasta que `i` sea menor que `10`, o sea `9`.
3. Incremento o Decremento: Debemos actualizar el contador, comúnmente incrementándolo o decrementándolo al final de cada iteración, por ejemplo `for(...; ...; i++)` tendrá incrementos de `1`.

Dicho lo anterior, la estructura completa de un ciclo `for` sería:

```
for (int i = 0; i < 10; i++) {
  // Bloque de código que se ejecuta mientras i sea menor que 10.
}
```

Este ciclo inicializa `i` con `0` y ejecuta el bloque de código siempre que `i` sea menor que `10`, incrementando `i` en `1` en cada iteración hasta que la condición ya no se cumpla, momento en el que termina el ciclo.

Ejemplo: Contar de 1 hasta 10

En el siguiente programa importamos el paquete `dart:io`, el cual permite el uso de funciones de entrada y salida (E/S), necesarias para imprimir texto en consola. En este caso

usamos la función `stdout.write()` para imprimir el valor actual de `i` seguido de un espacio. Esta función no agrega un salto de línea después de la salida, lo que significa que cada número se imprimirá seguido del otro sin empezar una nueva línea.

```
import ‘dart:io’;
void main() {
  for (var i = 1; i <= 10; i++) {
    stdout.write(“$i “);
  }
}
```

Salida:

```
1 2 3 4 5 6 7 8 9 10
```

Ejemplo: Imprimir elementos de tipo cadena (strings) de una lista

Para resolver este ejercicio, declaramos una lista llamada `municipios` que contiene nombres de ciudades. A través de un ciclo `for`, itera sobre la lista y para cada índice (`i`) desde `0` hasta el número de elementos en la lista menos uno, imprimiendo el elemento correspondiente de la lista.

```
void main() {
  var municipios = [
    “Colima”,
    “Villa de Álvarez”,
    “Comala”,
    “Coquimatlán”,
    “Manzanillo”
  ];
  for (var i = 0; i < municipios.length; i++) {
    print(municipios[i]);
  }
}
```

Salida:

```
Colima
Villa de Álvarez
Comala
Coquimatlán
Manzanillo
```

Estructuras repetitivas for anidadas

Estos ciclos se dan cuando un ciclo `for` se encuentra dentro de otro ciclo `for`. Esta estructura se utiliza comúnmente para recorrer datos multidimensionales, como matrices o tablas, donde necesitamos un ciclo para recorrer las filas y otro para las columnas. En cada iteración del ciclo externo, el ciclo interno se ejecuta completamente, permitiendo así procesar cada elemento de la estructura de datos. Este patrón es ampliamente utilizado en operaciones que requieren revisar o computar combinaciones de elementos, como la multiplicación de matrices o la búsqueda de datos en matrices bidimensionales.

Ejemplo: Generar tablas de multiplicar

El siguiente código imprimirá la tabla de multiplicar del 1 al 5. Para cada valor de `i` del 1 al 5, imprimirá su producto con cada valor de `j` del 1 al 5, la salida muestra las multiplicaciones de la forma 1 x 1 = 1 hasta 5 x 5 = 25.

```
void main() {
  for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 5; j++) {
      print(“$i x $j = ${i * j}”);
    }
  }
}
```

Estructura repetitiva while

El bucle o ciclo while es una estructura de control que permite ejecutar un bloque de código de manera repetida mientras se cumpla una condición especificada. Esta estructura, básicamente tiene los mismos elementos que vino en el ciclo `for`. A continuación, desglosa su comportamiento:

- Evaluación de la condición inicial: Antes de ejecutar el bloque de código dentro del `while`, la condición se evalúa. Si la condición se evalúa como falsa (`false`) desde el principio, el bloque de código dentro del `while` no se ejecutará ni una sola vez.
- Ejecución condicional: Para que la estructura `while` ejecute su bloque de código, la condición debe ser verdadera (`true`).
- Prevención de ciclos infinitos: Es importante que dentro del bloque de código del ciclo existan mecanismos para cambiar el estado de la condición a `false` eventualmente. De no ser así, si la condición siempre se evaluará como verdadera, y el programa entrará en un ciclo infinito, repitiendo el bloque de código indefinidamente y lo más probable es que cause efectos secundarios (que el programa se detenga o se comporte de manera no deseada).
- La sintaxis básica de un bucle `while` en Dart (y en la mayoría de los lenguajes de programación) es la siguiente:

```
while (condición) {
// Bloque de código a ejecutar mientras la condición sea verdadera
}
```

Ejemplo: Ciclo que se ejecuta sólo una vez

El siguiente ejemplo es un programa simple en Dart que utiliza un bucle `while` para imprimir valores basados en la variable booleana `isTrue`. Al inicio esta variable se establece en `true`, lo que significa que la condición del bucle `while` se cumple inicialmente y entra en el bucle. Después de que la primera iteración del bucle se ejecuta y se imprimen los valores, `isTrue` se establece en `false`, lo que hace que la condición del bucle `while` sea `false`, haciendo que el bucle se detenga y no se realicen más iteraciones. Por último, después de salir del bucle, el programa imprime nuevamente el valor de `isTrue`, que ahora es `false`.

```
void main() {
  bool isTrue = true;
  while (isTrue) {
    print(isTrue);
    isTrue = false;
  }
  print(isTrue);
}
```

Salida:

```
true
false
```

Ejemplo: Ciclo infinito

Algunos de los problemas comunes cuando se inicia en el mundo de la programación son los errores de lógica. El siguiente programa tiene un bucle `while` que pretende ejecutarse mientras la condición `isTrue` sea verdadera; sin embargo, hay un grave problema en la lógica: la condición `isTrue` nunca se modifica dentro del bucle `while`, lo que provoca un ciclo infinito. Para resolver este problema y evitar el ciclo infinito, necesitamos incluir una instrucción dentro del bucle que cambie la condición a `false` en algún momento como lo hicimos en el ejemplo anterior.

```
void main() {
  bool isTrue = true;
  while (isTrue) {
    print(isTrue);
  }
  print(isTrue);
}
```

Salida:

```
true
true
true
true
true
true
true
true
...
```

Ejemplo: Contar de 1 hasta 10

Este programa inicia la variable num en 1 y, en cada iteración del bucle, imprime el valor de num seguido de un espacio. Luego incrementa num en 1 con num++. El bucle continúa hasta que num sea mayor que 10, momento en el que la condición ya no se cumple y el ciclo termina.

```
import ‘dart:io’;

void main() {
  int num = 1;
  while (num <= 10) {
    stdout.write(“$num “);
    num++;
  }
}
```

Salida:

```
1 2 3 4 5 6 7 8 9 10
```

Ciclo do-while

El ciclo do-while es una estructura de control repetitiva que ejecuta un bloque de código al menos una vez y luego continúa ejecutándolo mientras la condición especificada se evalúe como verdadera. La principal característica que diferencia al ciclo do-while de un while es que la verificación de la condición se realiza al final de cada iteración, garantizando así que el cuerpo del ciclo se ejecute al menos una vez, incluso si la condición es falsa desde el inicio del ciclo.

Su sintaxis es:

```
do {
  // Código a ejecutar en cada iteración
} while (condicion);
```

Ejemplo: Ciclo que se ejecuta sólo una vez

```
void main() {
  bool isTrue = true;

  do {
    print(isTrue);
    isTrue = false;
  } while (isTrue);
  print(isTrue);
}
```

Salida:

```
true
false
```

Ejemplo: Ciclo infinito

```
void main() {
  bool isTrue = true;

  do {
    print(isTrue);
  } while (isTrue);
  print(isTrue);
}
```

Salida:

```
true
true
true
true
true
true
true
true
...
```

Ejemplo: Contar del 1 al 10

```
import ‘dart:io’;

void main() {
  int num = 1;
```

```
  do {
    stdout.write(“$num “);
    num++;
  } while (num <= 10);
}
```

Salida:

```
1 2 3 4 5 6 7 8 9 10
```

For in

- Se usa para estructuras iterables o colecciones (listas).

Ejemplo: Imprimir los elementos de una lista

```
import ‘dart:io’;

void main() {
  var municipios = [
    “Colima”,
    “Villa de Álvarez”,
    “Comala”,
    “Coquimatlán”,
    “Manzanillo”
  ];
  for (String m in municipios) {
    print(m);
  }
}
```

Salida:

```
Colima
Villa de Álvarez
Comala
Coquimatlán
Manzanillo
```

Ejemplo: Imprimir elementos enteros de una lista

```
import ‘dart:io’;

void main() {
  var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
```

```
  for (int i in nums) {
    stdout.write(“$i “);
  }
}
```

Salida:

```
1 2 3 4 5 6 7 8 9 10
```

Foreach

Ejemplo: Imprimir elementos cadena (strings) de una lista

```
void main() {
  var municipios = [
    “Colima”,
    “Villa de Álvarez”,
    “Comala”,
    “Coquimatlán”,
    “Manzanillo”
  ];

  municipios.forEach((e) {
    print(e);
  });
}
```

Salida:

```
Colima
Villa de Álvarez
Comala
Coquimatlán
Manzanillo
```

Ejemplo: Imprimir elementos enteros de una lista

```
import ‘dart:io’;

void main() {
  var nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  nums.forEach((e) {
    stdout.write(“$e “);
  });
}
```

Salida:

```
1 2 3 4 5 6 7 8 9 10
```

Generadores

Ejemplo: Contar del 1 al 10 mediante generadores

```
import ‘dart:io’;

void main() {
  var nums = List<int>.generate(10, (i) => i + 1);
  for (int i in nums) {
    stdout.write(“$i “);
  }
}
```

Reductores (reduce)

Ejemplo: Obtener la suma de los números enteros de los primeros 10 números naturales mediante for

```
void main(List<String> args) {
  int suma = 0;
  for (var i = 1; i < 11; i++) {
    suma += i;
  }
  print(suma);
}
```

Salida:

```
55
```

Ejemplo: Obtener la suma de los números enteros de los primeros 10 números naturales mediante forEach

```
void main() {
  var nums = List<int>.generate(10, (i) => i + 1);
  var sum = 0;
  nums.forEach((i) => sum += i);
  print(sum);
}
```

Salida:

```
55
```

Ejemplo: Obtener la suma de los números enteros de los primeros 10 números naturales mediante reduce

```
void main() {
  var nums = List<int>.generate(10, (i) => i + 1);
  var sum = nums.reduce((a, b) => a + b);
  print(sum);
}
```

Salida:

```
55
``


```java
void main() {
  var nums = List<int>.generate(10, (i) => i + 1);
  var sum = 0;
  nums.forEach((i) => sum += i);
  print(sum);
}
```

break y continue

- break rompe la estructura repetitiva o condicional (salida forzada).
- continue avanza al siguiente valor de la estructura repetitiva sin romper el ciclo (pausa).

Ejemplo: Ciclo de 1 a 10 que se rompe en 5 (break)

```
void main() {
  for (var i = 1; i <= 10; i++) {
    if (i > 5) {
      break;
    }
    print(i);
  }
}
```

Salida:

```
1
2
3
4
5
```
```

Ejemplo: Ciclo de 1 a 10 que sólo imprime los valores mayores a 5 (continue)

```
void main() {
  for (var i = 1; i <= 10; i++) {
    if (i > 5) {
      print(i);
    } else {
      continue;
    }
  }
}
```

Salida:

```
6
7
8
9
10
```

Ejemplo: Ejemplos con labels

```
void main() {
  // Ejemplo 1: Buscar un número en una matriz
  print('=== Ejemplo 1: Búsqueda en matriz ===');
  List<List<int>> matriz = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12],
    [13, 14, 15, 16]
  ];

  int objetivo = 11;
  bool encontrado = false;

  busqueda:  // Label para el bucle externo
  for (int i = 0; i < matriz.length; i++) {
    for (int j = 0; j < matriz[i].length; j++) {
      if (matriz[i][j] == objetivo) {
        print('Número $objetivo encontrado en posición [$i[$j]');
        encontrado = true;
        break búsqueda;  // Sale de ambos bucles
      }
    }
```

```
}

if (!encontrado) {
  print('Número $objetivo no encontrado');
}

// Ejemplo 2: Validar credenciales con intentos
print('\n=== Ejemplo 2: Sistema de login ===');
List<Map<String, String>> usuarios = [
  {'user': 'admin', 'pass': '1234'},
  {'user': 'guest', 'pass': 'guest'},
  {'user': 'user1', 'pass': 'pass1'}
];

String usuarioBuscado = 'guest';
String passwordIngresado = 'guest';

autenticacion:  // Label
for (var usuario in usuarios) {
  if (usuario['user'] == usuarioBuscado) {
    if (usuario['pass'] == passwordIngresado) {
      print('✓ Login exitoso para $usuarioBuscado');
      break autenticacion;
    } else {
      print('✗ Contraseña incorrecta');
      break autenticacion;
    }
  }
}

// Ejemplo 3: Continue con labels
print('\n=== Ejemplo 3: Procesar datos con saltos ===');

procesamientoExterno:
for (int categoria = 1; categoria <= 3; categoria++) {
  print('\nCategoría $categoria:');

  for (int item = 1; item <= 5; item++) {
    // Saltar toda la categoría 2
    if (categoria == 2 && item == 1) {
      print('  Categoría 2 omitida completamente');
```

```
        continue procesamientoExterno;
      }

      // Saltar items pares
      if (item % 2 == 0) {
        continue;
      }

      print('  Procesando item $item');
    }
  }

  // Ejemplo 4: Juego de adivinanza
  print('\n=== Ejemplo 4: Búsqueda de combinación ===');

  int codigoSecreto1 = 3;
  int codigoSecreto2 = 7;

  buscarCódigo:
  for (int i = 0; i <= 10; i++) {
    for (int j = 0; j <= 10; j++) {
      int intentos = i * 10 + j + 1;

      if (i == codigoSecreto1 && j == codigoSecreto2) {
        print('🎉 ¡Código encontrado! [$i, $j]');
        print('Total de intentos: $intentos');
        break buscarCodigo;
      }
    }
  }
}
```

Salidas:

```
=== Ejemplo 1: Búsqueda en matriz ===
Buscando en fila 0...
Buscando en fila 1...
Buscando en fila 2...
Número 11 encontrado en posición [2][2]

=== Ejemplo 2: Sistema de login ===
✓ Login exitoso para guest
```

```
=== Ejemplo 3: Procesar datos con saltos ===

Categoría 1:
  Procesando item 1
  Procesando item 3
  Procesando item 5

Categoría 2:
  Categoría 2 omitida completamente

Categoría 3:
  Procesando item 1
  Procesando item 3
  Procesando item 5

=== Ejemplo 4: Búsqueda de combinación ===
🎉 ¡Código encontrado! [3, 7]
Total de intentos: 38
```

Explicación de los ejemplos

1. **Búsqueda en matriz**: Usa un label busqueda: para salir de ambos bucles cuando encuentra el número objetivo.
2. **Sistema de login**: Usa autenticacion: para salir del bucle una vez que valida las credenciales (correctas o incorrectas).
3. **Continue con labels**: Muestra cómo **continue procesamientoExterno** salta a la siguiente iteración del bucle externo, omitiendo todo el procesamiento interno.
4. **Búsqueda de combinación**: Simula encontrar un código secreto y sale inmediatamente con break **buscarCodigo**.

Cuándo usar labels

☑ Bucles anidados donde necesitas salir de varios niveles.
☑ Búsquedas en estructuras multidimensionales.
☑ Control de flujo complejo con múltiples condiciones.
❌ Evita usarlos en exceso, pueden dificultar la lectura del código.

Sentencia assert

En el contexto de la programación en Dart, el tema “assert” reviste una importancia vital en el aseguramiento de la calidad y corrección de un programa. Un “assert” es una sentencia que permite verificar condiciones específicas dentro del código y garantizar que estas condiciones sean verdaderas en un punto determinado del programa. Si una afirmación (assertion) resulta falsa, el programa interrumpirá su ejecución y mostrará información relevante para ayudar a identificar y corregir el problema.

Esta herramienta es crucial durante el desarrollo y mantenimiento del software, ya que brinda a los desarrolladores una forma efectiva de atrapar errores y verificar suposiciones en el código. Los “asserts” proporcionan una manera estructurada de verificar que el comportamiento esperado se cumple, lo que resulta en un código más confiable y robusto.

A lo largo de esta sección, exploraremos el uso y la aplicación de “assert” en Dart, así como ejemplos prácticos que ilustrarán cómo implementar afirmaciones efectivas para asegurar la integridad y confiabilidad de nuestro código, a continuación, te muestro las acciones básicas que realiza assert:

- Validar si una variable cumple con una regla específica.
- Se usa en el desarrollo, no en el despliegue de la aplicación.

Sintaxis:

```
assert(rule);
```

Ejemplo: Validar si un número es mayor que 5

```
void main() {
  int variable = 10;
  assert(variable > 5);
  //print(“Validación correcta”);
}
```

- La ejecución se realiza activando una bandera:

```
C:\mi_app>  dart --enable-asserts run
```

Salida:

```
C:\mi_app> dart --enable-asserts run
Building package executable...
Built mi_app:mi_app.
```

Lo cual significa que pasó la prueba.
Si cambiamos el valor de variable para que no cumpla, por ejemplo estableciéndole un 4, la salida cambiaría a:

```
C:\mi_app> dart --enable-asserts run
Building package executable...
```

```
Built mi_app:mi_app.
Unhandled exception:
‘file:///D:/wkspaceLearnDart/mi_app/bin/mi_app.dart’: Failed as-
sertion: line 14 pos 10: ‘variable > 5’: is not true.
#0      _AssertionError._doThrowNew (dart:core-patch/errors_
patch.dart:51:61)
#1      _AssertionError._throwNew (dart:core-patch/errors_patch.
dart:40:5)
#2      main (file:///D:/wkspaceLearnDart/mi_app/bin/mi_app.
dart:14:10)
#3      _delayEntrypointInvocation.<anonymous closure> (dart:iso-
late-patch/isolate_patch.dart:297:19)
#4      _RawReceivePortImpl._handleMessage (dart:isolate-patch/
isolate_patch.dart:192:12)
```

Indicar el número de línea donde se presentó la falla en la prueba

Leer datos de la consola

En esta sección, exploraremos la importante tarea de leer datos desde la consola en el contexto de la programación en Dart. La interacción con el usuario es un aspecto fundamental en muchas aplicaciones, y saber cómo recibir y procesar datos ingresados por el usuario es esencial para crear programas interactivos y útiles.

Aprenderemos cómo solicitar información al usuario desde la consola mediante ejemplos prácticos. Desde pedir el nombre de una persona hasta calcular su edad con base en el año o mes de nacimiento, abordaremos distintos escenarios comunes que involucran entrada de datos. Además, exploraremos cómo usar la librería DateTime para operaciones relacionadas con fechas y edades, lo que añade una dimensión adicional de complejidad y utilidad a nuestras aplicaciones.

Continuamos mostrando ejemplos detallados que ilustrarán cómo leer datos de la consola de manera efectiva en Dart y cómo utilizar esa información para realizar tareas útiles en nuestros programas.

Ejemplo: Preguntar el nombre desde el teclado

```
import ‘dart:io’;

void main(List<String> args) {
  stdout.write(“¿Cuál es tu nombre?\n”);
  String? nombre = stdin.readLineSync();
```

```
  print(“Hola $nombre”);
}
```

Salida:

```
¿Cuál es tu nombre?
Walter
Hola Walter
```

Ejemplo: Calcular la edad de una persona de acuerdo al año de nacimiento

```
import ‘dart:io’;

void main(List<String> args) {
  stdout.writeln(“¿En qué año naciste?”);
  int aNacimiento = int.parse(stdin.readLineSync()!);

  print(“Tienes ${2023 - aNacimiento} años”);
}
```

Salida:

```
¿En qué año naciste?
1971
Tienes 52 años
```

Ejemplo: Calcular la edad de una persona de acuerdo al año de nacimiento usando la librería DateTime

```
import ‘dart:io’;

void main(List<String> args) {
  DateTime hoy = DateTime.now();

  stdout.writeln(“¿En qué año naciste?”);
  int aNacimiento = int.parse(stdin.readLineSync()!);

  print(“Tienes ${hoy.year - aNacimiento} años”);
}
```

Salida:

```
¿En qué año naciste?
1971
Tienes 51 años
```

Ejemplo: Calcular la edad de una persona de acuerdo al mes y año de nacimiento

```
import ‘dart:io’;

void main(List<String> args) {
  DateTime hoy = DateTime.now();

  stdout.writeln(“¿En qué año naciste?”);
  int aNacimiento = int.parse(stdin.readLineSync()!);

  stdout.writeln(“¿En qué número de mes ?”);
  int mNacimiento = int.parse(stdin.readLineSync()!);

  if (mNacimiento <= hoy.month) {}
  print(“Tienes ${hoy.year - aNacimiento} años”);
}
```

Salida:

```
¿En qué año naciste?
1971
¿En qué número de mes ?
11
Tienes 51 años
```

Ejemplo: Calculadora simple

```
import ‘dart:io’;

void main() {
  print(“Dame un número entero:”);
  int? num1 = int.parse(stdin.readLineSync()!);
  print(“Dame un otro entero:”);
  int? num2 = int.parse(stdin.readLineSync()!);
  print(“Escribe la operación a realizar [+,-,*,/]”);
  String? op = stdin.readLineSync();
  switch (op) {
    case ‘+’:
      print(“$num1 $op $num2=${num1 + num2}”);
      break;
    case ‘-’:
      print(“$num1 $op $num2=${num1 - num2}”);
      break;
    case ‘*’:
```

```
      print(“$num1 $op $num2=${num1 * num2}”);
      break;
    case ‘/’:
      print(“$num1 $op $num2=${num1 / num2}”);
      break;
    default:
      print(“La operación $op no se puede realizar”);
  }
  print(num1);
}
```

Capítulo 5
Colecciones

Las colecciones son fundamentales en la programación, permitiendo agrupar y gestionar conjuntos de datos de diversas formas. En Dart, disponemos de varias estructuras de colecciones que nos permiten organizar la información de manera eficiente y efectiva.

A lo largo de los temas que abordaremos, comprenderemos cómo trabajar con listas, conjuntos y mapas, así como sus diferentes variantes y operaciones asociadas. Desde la inicialización de colecciones hasta su manipulación y utilización en escenarios específicos, exploraremos ejemplos detallados que ilustrarán cada concepto y nos permitirán aplicarlos en nuestros programas.

Conocer a fondo las colecciones es esencial para cualquier programador, ya que nos brindan herramientas poderosas para manejar datos y resolver problemas de manera estructurada y organizada.

Listas

- Grupo de elementos con características comunes entre sí.

Ejemplo: Tipado inferido

```
void main(List<String> args) {
  var miLista1 = [1, 2, 3, 4, 5]; //mismo tipo de dato
  print(miLista1);

  var miLista2 = [‘uno’, ‘dos’, ‘tres’, ‘cuatro’, ‘cinco’]; //
mismo tipo de dato
  print(miLista2);

  var miLista3 = [1, ‘uno’, 2, ‘dos’]; //distinto tipo de dato
  print(miLista3);
}
```

Ejemplo: Tipado estático vs dinámico

```
void main(List<String> args) {
  List<int> miLista1 = [1, 2, 3, 4, 5]; //mismo tipo de dato
  print(miLista1);
```

```
  List<String> miLista2 = [
    ‘uno’,
    ‘dos’,
    ‘tres’,
    ‘cuatro’,
    ‘cinco’
  ]; //mismo tipo de dato
  print(miLista2);

  List<dynamic> miLista3 = [1, ‘uno’, 2, ‘dos’]; //distinto tipo
de dato
  print(miLista3);
}
```

Ejemplo: Usando el constructor de listas

```
void main(List<String> args) {
  // ignore: prefer_collection_literals
  var miLista = List.empty(growable: true);

  miLista.add(‘uno’);
  miLista.add(‘dos’);
  miLista.add(‘tres’);
  miLista.add(‘cuatro’);
  miLista.add(‘cinco’);
  print(miLista);
}
```

Salida:

```
[uno, dos, tres, cuatro, cinco]
```

Ejemplo: Eliminar un elemento de la lista

```
void main(List<String> args) {
  var miLista = [‘uno’, ‘dos’, ‘tres’, ‘cuatro’, ‘cinco’];
  print(miLista);
  miLista.remove(‘tres’);
  print(miLista);
}
```

```
[uno, dos, tres, cuatro, cinco]
[uno, dos, cuatro, cinco]
```

Ejemplo: Eliminar un elemento de la lista en determinada posición

```
void main(List<String> args) {
  var miLista = [‘uno’, ‘dos’, ‘tres’, ‘cuatro’, ‘cinco’];
  print(miLista);
  miLista.removeAt(2);
  print(miLista);
}
```

Salida:

```
[uno, dos, tres, cuatro, cinco]
[uno, dos, cuatro, cinco]
```

Indexación de listas

```
void main(List<String> args) {
  List<String> miLista = [
    ‘uno’,
    ‘dos’,
    ‘tres’,
    ‘cuatro’,
    ‘cinco’
  ]; //mismo tipo de dato
  print(miLista[0]);
  print(miLista[1]);
  print(miLista[2]);

  for (var i = 0; i < miLista.length; i++) {
    print(miLista[i]);
  }

  miLista.forEach((element) {
    print(element);
  });
  miLista.forEach((element) => print(element));

  print(“----”);
  miLista.forEach((print));
}
import ‘dart:io’;

void main() {
//listas
```

```
  final miLista = const [1, 2, 3, 4];
  print(miLista);
  for (var i = 0; i < miLista.length; i++) {
    stdout.write("$i: ${miLista[i]}");
  }
  print("");
  stdout.write("\n");
  for (var i = 0; i < miLista.length; i++) {
    print("$i: ${miLista[i]}");
  }
  miLista.forEach((element) {
    print(element);
  });
  miLista.forEach((element) => print(element));
}
```

Listas fijas

```
void main() {
  List<int> fixed = List.filled(10, 0);
  print(fixed);
  for (var i = 0; i < fixed.length; i++) {
    fixed[i] = i;
  }
  print(fixed);
  fixed.forEach((element) {
    print(element);
  });
}
```

Ejemplo: Listas dinámicas (crecen)

```
void main() {
  List<int> listaDinamica = [];
  for (var i = 1; i <= 10; i++) {
    listaDinamica.add(i);
  }
  print(listaDinamica);
  listaDinamica.addAll([11, 12, 13, 14, 15]);
  print(listaDinamica);
}
```

Salida:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Ejemplo: Copia de listas

```
void main() {
  var listaOriginal = [1, 2, 3, 4, 5];
  var listaClone = listaOriginal;
  print(“Lista Original: $listaOriginal”);
  print(“Lista Clone   : $listaClone”);
}
```

Salida:

```
Lista Original: [1, 2, 3, 4, 5]
Lista Clone   : [1, 2, 3, 4, 5]
```

Ejemplo: Copia de listas con cambio de valores

```
void main() {
  var listaOriginal = [1, 2, 3, 4, 5];
  var listaClone = listaOriginal;
  listaClone[0] = 23;
  print(“Lista Original: $listaOriginal”);
  print(“Lista Clone   : $listaClone”);
}
```

Salida:

```
Lista Original: [23, 2, 3, 4, 5]
Lista Clone   : [23, 2, 3, 4, 5]
```

Ejemplo: Clonación de listas

```
void main() {
  var listaOriginal = [1, 2, 3, 4, 5];
  var listaClone = listaOriginal.map((num) => num).toList();
  print(“Lista Original: $listaOriginal”);
  print(“Lista Clone   : $listaClone”);
}
```

Salida:

```
Lista Original: [1, 2, 3, 4, 5]
Lista Clone   : [1, 2, 3, 4, 5]
```

Ejemplo: Clonación de listas con cambio de valores

```
void main() {
  var listaOriginal = [1, 2, 3, 4, 5];
  var listaClone = listaOriginal.map((num) => num).toList();
  listaClone[0] = 23;
  print(“Lista Original: $listaOriginal”);
  print(“Lista Clone   : $listaClone”);
}
```

Salida:

```
Lista Original: [1, 2, 3, 4, 5]
Lista Clone   : [23, 2, 3, 4, 5]
```

Ejemplo: Clonación de listas (alternativa)

```
void main() {
  var listaOriginal = [1, 2, 3, 4, 5];
  var listaClone = []..addAll(listaOriginal);
  listaClone[0] = 23;
  print(“Lista Original: $listaOriginal”);
  print(“Lista Clone   : $listaClone”);
}
```

Salida:

```
Lista Original: [1, 2, 3, 4, 5]
Lista Clone   : [23, 2, 3, 4, 5]
```

Ejemplo: Clonación de listas (alternativa) con cambio de valor

```
void main() {
  var listaOriginal = [1, 2, 3, 4, 5];
  var listaClone = List.from(listaOriginal);
  listaClone[0] = 23;
  print(“Lista Original: $listaOriginal”);
  print(“Lista Clone   : $listaClone”);
}
```

Salida:

```
Lista Original: [1, 2, 3, 4, 5]
Lista Clone   : [23, 2, 3, 4, 5]
```

Ejemplo: Operador spread

```
void main() {
  List<String> personas = [“Hugo”, “Paco”, “Luis”];

  var nombres = [...personas];

  print(personas);
  print(nombres);

  nombres[1] = “María”;
  print(personas);
  print(nombres);
}
```

Salida:

```
[Hugo, Paco, Luis]
[Hugo, Paco, Luis]
[Hugo, Paco, Luis]
[Hugo, María, Luis]
```

Ejemplo: Eliminar un elemento

```
void main() {
  var miLista = [1, “rojo”, true, 3.1416, “rojo”];
  miLista.remove(“rojo”);
  print(miLista);
}
```

Salida:

```
[1, true, 3.1416, rojo]
```

Ejemplo: Eliminar un elemento todas las veces que se encuentra repetido

```
void main() {
  var miLista = [1, “rojo”, true, 3.1416, “rojo”];
  miLista.removeWhere((element) => element == “rojo”);
  print(miLista);
}
```

Salida:

```
[1, true, 3.1416]
```

Ejemplo: Eliminar un elemento por su posición

```
void main() {
  var miLista = [1, “rojo”, true, 3.1416, “rojo”];
  miLista.removeAt(2);
  print(miLista);
}
```

Salida:

```
[1, rojo, 3.1416, rojo]
```

Ejemplo: Eliminar todos los pares de una lista de números (for, for in)

```
void main() {
  var miLista = [1, 5, 6, 10, 5, 6, 2, 18, 45, 36];

/*
  for (var element in miLista) {
    if (element % 2 == 0) {
      miLista.remove(miLista.indexOf(element));
    }
  }
  print(miLista); */

  /* for (var i = 0; i < miLista.length; i++) {
    if (miLista[i] % 2 == 0) {
      miLista.removeAt(i);
    }
  }
  print(miLista);
 */

  /*
  var tam = miLista.length;
  for (var i = 0; i < tam; i++) {
    if (miLista[i] % 2 == 0) {
      miLista.removeAt(i);
    }
  }
  print(miLista); */
}
```

Ejemplo: Eliminar todos los pares de una lista de números (for in)

```
void main() {
  var miLista = [1, 5, 6, 10, 5, 6, 2, 18, 45, 36];
  List<int> listaImpares = [];

  for (var element in miLista) {
    if (element % 2 != 0) {
      //miLista.remove(element);
      listaImpares.add(element);
    }
  }
  print(miLista);
  print(listaImpares);
}
```

Salida:

```
[1, 5, 6, 10, 5, 6, 2, 18, 45, 36]
[1, 5, 5, 45]
```

Ejemplo: Eliminar todos los pares de una lista de números (método removeWhere)

```
void main() {
  var miLista = [1, 5, 6, 10, 5, 6, 2, 18, 45, 36];

  miLista.removeWhere((element) => element % 2 == 0);
  print(miLista);
}
```

Salida

```
[1, 5, 5, 45]
```

Ejemplo: Eliminar elementos repetidos en una lista

```
void main() {
  var listaNumerosRepetidos = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5];
  var listaNumeros = listaNumerosRepetidos.toSet();

  print("Lista: $listaNumeros");
  print("Lista: ${listaNumeros.toList()}");
}
```

Salida:

```
Lista: {1, 2, 3, 4, 5}
Lista: [1, 2, 3, 4, 5]
```

Ejemplo:

```
void main() {
  var listaNumerosRepetidos = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5];
  var listaNumeros = listaNumerosRepetidos.toSet().toList();

  print("Lista: $listaNumeros");
}
```

Salida:

```
Lista: [1, 2, 3, 4, 5]
```

Ejemplo: Obtener una cantidad de elementos de una lista

```
void main() {
  var listaNumeros = [1, 2, 3, 4, 5, 6];
  print(listaNumeros.take(4));
  print(listaNumeros.take(4).toList());
}
```

Salida:

```
(1, 2, 3, 4)
[1, 2, 3, 4]
```

Ejemplo:

```
void main() {
  var listaNumeros = [1, 2, 3, 4, 5, 6];
  print(listaNumeros.skip(4));
  print(listaNumeros.skip(4).toList());
}
```

Salida:

```
(5, 6)
[5, 6]
```

Ejemplo:

```
void main() {
  var listaNumeros = [1, 2, 3, 4, 5, 6];
  print(listaNumeros.take(listaNumeros.length ~/ 2).skip(2).
toList());
}
```

Salida:

```
[3]
```

Inicializar una lista con n elementos con valores aleatorios.

```
import ‘dart:math’;

void main() {
  List<int> list1 = List.filled(10, 0);
  var list2 = List<int>.filled(10, 0);
  for (var i = 0; i < list1.length; i++) {
    list1[i] = Random().nextInt(20);
    list2[i] = Random().nextInt(30);
  }
  print(“list1: $list1”);
  print(“list2: $list2”);
  var list3 = [...list1, ...list2];
  print(“list3: $list3”);
  Set<int> set1 = {};
  for (var element in list3) {
    set1.add(element);
  }
  print(“set1: $set1”);
}
```

Conjuntos (sets)

- Usa la palabra reservada Set.
- No se pueden almacenar valores duplicados.
- Se puede utilizar para eliminar elementos repetidos en una lista.

Ejemplo: Tipado inferido

```
void main(List<String> args) {
  var miSet1 = {1, 2, 3, 4, 5}; //mismo tipo de dato
  print(miSet1);

  var miSet2 = {‘uno’, ‘dos’, ‘tres’, ‘cuatro’, ‘cinco’}; //mismo
tipo de dato
  print(miSet2);

  var miSet3 = {1, ‘uno’, 2, ‘dos’}; //distinto tipo de dato
  print(miSet3);
}
```

Tipado estático:

```
void main(List<String> args) {
  Set<int> miSet1 = {1, 2, 3, 4, 5}; //mismo tipo de dato
  print(miSet1);

  Set<String> miSet2 = {
    ‘uno’,
    ‘dos’,
    ‘tres’,
    ‘cuatro’,
    ‘cinco’
  }; //mismo tipo de dato
  print(miSet2);

  Set<dynamic> miSet3 = {1, ‘uno’, 2, ‘dos’}; //distinto tipo de
dato
  print(miSet3);

  Set<int> miSet4 = Set<int>();
  miSet4.add(1);
  miSet4.add(2);
  miSet4.add(3);
  miSet4.add(4);
  miSet4.add(5);
  print(miSet4);

  var miSet5 = <String>{‘Hugo’, “Paco”, “Luis”};
  miSet5.forEach((element) {
    print(element);
  });
}

  Set miSet6 = Set();
  miSet6.add(“Hugo”);
  miSet6.add(10);
  miSet6.add(true);
  miSet6.add(3.1416);
  print(miSet6);
```

Salida:

```
{1, 2, 3, 4, 5}
{uno, dos, tres, cuatro, cinco}
{1, uno, 2, dos}
{1, 2, 3, 4, 5}
Hugo
Paco
Luis
{Hugo, 10, true, 3.1416}
```

Ejemplo: Diferencia entre Listas y Sets

```
void main(List<String> args) {
  List<int> miList1 = [1, 2, 3, 4, 5];
  Set<int> miSet1 = {1, 2, 3, 4, 5};

  print(miList1);
  print(miSet1);

  miList1.add(6);
  miSet1.add(6);
  print(miList1);
  print(miSet1);

  List<int>.generate(5, (i) => i + 1).forEach((element) =>
miList1.add(6));
  print(miList1);
  List<int>.generate(5, (i) => i + 1).forEach((element) =>
miSet1.add(6));
  print(miSet1);
}
```

Set sin valores

```
void main() {
  Set numeros = Set();
  numeros.add(1);
  numeros.add(2);
  numeros.add(2);
  print(numeros);
}
```

Salida:

```
{1, 2}
```

Recorrer los elementos del set

Ejemplo:

```
void main() {
  Set numeros = Set();
  numeros.add(1);
  numeros.add(2);
  numeros.add(2);
  numeros.forEach((numero) {
    print(numero);
  });
}
```

Salida:

```
1
2
```

Eliminar elementos repetidos de una lista

```
void main() {
  List<int> numeros = [];
  numeros.add(1);
  numeros.add(2);
  numeros.add(2);
  numeros.add(3);
  numeros.add(3);
  print(numeros);

  Set<int> numerosSet = numeros.toSet();
  print(numerosSet);
}
```

Convertir un set a list

```
void main() {
  List<int> numeros = [];
  numeros.add(1);
  numeros.add(2);
  numeros.add(2);
```

```
  numeros.add(3);
  numeros.add(3);
  print(numeros);

  Set<int> numerosSet = numeros.toSet();
  print(numerosSet);

  numeros = numerosSet.toList();
  print(numeros);
}
```

Convertir un set a list con map

```
void main() {
  Set numeros = Set();
  numeros.add(1);
  numeros.add(2);
  numeros.add(2);
  var listNums = numeros.map((numero) => numero).toList();
  print(listNums);
}
```

Salida:

```
[1, 2]
```

Ejemplo:

```
void main() {
  Set numeros = Set();
  numeros.add(1);
  numeros.add(2);
  numeros.add(2);
  print(numeros.toList());
  print(numeros);
}
```

Salida:

```
[1, 2]
{1, 2}
```

Ejemplo:

```
import ‘dart:math’;

void main() {
  Set<int> setNumeros = Set();

  var num = Random();
  for (var i = 1; i <= 10; i++) {
    setNumeros.add(num.nextInt(10));
  }
  print(setNumeros);
  int cont = 1;
  for (var item in setNumeros) {
    print(“Número $cont: $item”);
    cont++;
  }
  if (setNumeros.contains(4)) {
    print(“Se generó el número 4”);
  } else {
    print(“No se generó el No. 4”);
  }
  print(setNumeros.remove(3)); //si se encuentra es true, si no
false
}
void main() {
  List numeros = [5, 6, 7, 4, 5, 6, 7, 5, 5, 4];
  Set setNumeros = Set();
  setNumeros.addAll(numeros);
  print(numeros);
  print(setNumeros);
}
void main() {
  List numeros = [5, 6, 7, 4, 5, 6, 7, 5, 5, 4];
  Set setNumeros = Set.from(numeros);
  print(numeros);

  print(setNumeros);
}
```

Maps

- Permite almacenar datos con una clave o llave.
- También se conocen como diccionarios.
- Es una estructura par llave-valor.

Ejemplo: Map de forma directa

```
void main() {
  var miMap = {“llave1”: “valor1”, “llave2”: 2, 3: “valor3”};
  print(miMap);
}
```

Salida:

```
{llave1: valor1, llave2: 2, 3: valor3}
```

Ejemplo: Map con su constructor

```
void main() {
  var miMap = Map();
  miMap[“llave1”] = “valor1”;
  miMap[“llave2”] = 2;
  miMap[3] = 3;
  print(miMap);
}
```

Salida:

```
{llave1: valor1, llave2: 2, 3: 3}
```

Ejemplo: Forma recomendable

```
void main() {
  var miMap = <dynamic, dynamic>{};
  miMap[“llave1”] = “valor1”;
  miMap[“llave2”] = 2;
  miMap[3] = 3;
  print(miMap);
}
```

Salida:

```
{llave1: valor1, llave2: 2, 3: 3}
```

Ejemplo: Forma recomendable con tipado

```
void main() {
  Map<dynamic, dynamic> miMap = <dynamic, dynamic>{};
  miMap[“llave1”] = “valor1”;
  miMap[“llave2”] = 2;
  miMap[3] = 3;
  print(miMap);
}
```

Salida:

```
{llave1: valor1, llave2: 2, 3: 3}
```

Ejemplo: Obtener las llaves y los valores de un map

```
void main() {
  var paises = {“USA”: 1, “MEXICO”: 2, “England”: 3};
  print(paises.keys);
  print(paises.values);
}
```

Salida:

```
(USA, MEXICO, England)
(1, 2, 3)
```

Ejemplo: Obtener las llaves y los valores de un map con for-in

```
void main() {
  var paises = {“USA”: 1, “MEXICO”: 2, “England”: 3};
  for (var keyPais in paises.keys) {
    print(“Clave: $keyPais”);
  }
  for (var valuePais in paises.values) {
    print(“Clave: $valuePais”);
  }
}
```

Salida:

```
Clave: USA
Clave: MEXICO
Clave: England
Clave: 1
Clave: 2
Clave: 3
```

Ejemplo: Obtener las llaves y los valores de un map con forEach

```
void main() {
  var paises = {“USA”: 1, “MEXICO”: 2, “England”: 3};
  paises.forEach(
    (key, value) {
      print(“Clave: $key, Valor: $value”);
    },
  );
}
```

Salida:

```
Clave: USA, Valor: 1
Clave: MEXICO, Valor: 2
Clave: England, Valor: 3
```

Ejemplo: Obtener las llaves y los valores de un map con forEach y función fat arrow

```
void main() {
  var paises = {“USA”: 1, “MEXICO”: 2, “England”: 3};
  paises.forEach((key, value) => print(“Clave: $key, Valor: $val-
ue”));
}
```

Salida:

```
Clave: USA, Valor: 1
Clave: MEXICO, Valor: 2
Clave: England, Valor: 3
```

Ejemplo: Obtener el valor de una llave

```
void main() {
  var paises = {“USA”: 1, “MEXICO”: 2, “England”: 3};
  print(paises);
  print(“${paises[‘MEXICO’]}”); //Acceso al valor
}
```

Salida:

```
{USA: 1, MEXICO: 2, England: 3}
2
```

Ejemplo: Obtener las llaves de un map con tipado estático

```
void main() {
  Map<String, int> paises = {“USA”: 1, “MEXICO”: 2, “England”:
```

```
3};
  for (var pais in paises.keys) {
    print(“Clave: $pais”);
  }
  print(paises);
  print(“${paises[‘Colima’]}”); //no existe: null
}
```

Ejemplo: Inicialización de un map

```
void main() {
  Map<String, int> paises = Map();
  //var paises = new Map();
  //Map<String, int> paises = {};
  //var paises = Map();
  paises = {“USA”: 1, “MEXICO”: 2, “England”: 3};
  for (var pais in paises.keys) {
    print(“Clave: $pais”);
  }
  print(paises);
  print(“${paises[‘MEXICO’]}”);
}
```

Ejemplo: Agregar un elemento al map

```
void main() {
  var paises = Map();
  paises = {“USA”: 1, “MEXICO”: 2, “England”: 3};
  for (var pais in paises.keys) {
    print(“Clave: $pais”);
  }
  print(paises);
  paises[“Colombia”] = 4;
  print(paises);
}
```

Ejemplo: Agregar un elemento al map

```
void main(List<String> args) {
  Map<String, int> valores = <String, int>{};

  valores[“uno”] = 1;
  print(valores);
}
```

Salida:

```
{uno: 1}
void main() {
  var paises = Map();
  paises = {“USA”: 1, “MEXICO”: 2, “England”: 3};
  for (var pais in paises.keys) {
    print(“Clave: $pais”);
  }
  print(paises);
  paises[“Colombia”] = 4;
  print(paises);
}
```

Ejemplo: Agregar varios elementos al map

```
void main() {
  var paises = new Map();
  paises = {“USA”: 1, “MEXICO”: 2, “England”: 3};

  paises.addAll({
    “Colombia”: 4,
    “Brasil”: 5,
    “Argentina”: 6,
  });

  print(paises);
}
```

Ejemplos:

```
void main() {
  var myList1 = [];
  var myList2 = <int>[];
  print(“$myList1, $myList2”);
  myList1.add(5);
  myList1.add(“hola”);
  print(myList1);
  myList2.add(5);
// myList2.add(“hola”);
  print(myList2);
}
void main() {
```

```
  var mySet1 = <dynamic>{};
  var mySet2 = <String>{};

  print("$mySet1, $mySet2");
  mySet1.add(5);
  mySet2.add("hola");
  print("$mySet1, $mySet2");
  mySet1.add("Mundo");
  print("$mySet1");
}
void main() {
  var myMap1 = {"uno": 1};
  var myMap2 = <String, int>{};
  print("$myMap1, $myMap2");
  var entrada = {"dos": 2};
  myMap1.addEntries(entrada.entries);
  print(myMap1);
  myMap2.addEntries(entrada.entries);
  print("$myMap1, $myMap2");
}
void main() {
  var listaInicial = [1, 2, 3];
  var listaFinal = [4, 5, 6];
  var listaTotal1 = [];
  listaTotal1.addAll(listaInicial);
  listaTotal1.addAll(listaFinal);
  print(listaTotal1);
  var listaTotal2 = [listaInicial, listaFinal];
  print(listaTotal2);
  var listaTotal3 = [...listaInicial, ...listaFinal];
  print(listaTotal3);
}
```

Spread Operator

```
void main() {
  var Map1 = {'uno': 1};
  var Map2 = {'dos': 2};
  var MapTotal = {...Map1, ...Map2};
  print(MapTotal);
  var set1 = {1, 2, 3};
  var set2 = {3, 4, 5};
```

```
  var setTotal = {...set1, ...set2};
  print(setTotal);
}
void main() {
  List numeros = [8, 13];
  numeros.add(3);
  numeros.add(9);
  for (var num in numeros) {
    print(num);
  }
  for (var i = 0; i < numeros.length; i++) {
    print(numeros[i]);
  }
}
```

Agregar elementos a un map

```
void main() {
  Map<String, dynamic> animales = {};
  animales[“leon”] = ‘ruge’;
  animales[“perro”] = ‘ladra’;
  animales[“vaca”] = ‘muge’;

  print(animales.keys);
  print(animales.values);
  for (var element in animales.entries) {
    print(‘${element.key}: ${element.value}’);
  }
}
```

Ejemplo: HashMaps

- No mantienen el orden de inserción.
- Maneja los datos de manera más eficiente que los maps.

```
import ‘dart:collection’;

void main() {
  HashMap animales = HashMap();

  animales[“leon”] = ‘ruge’;
  animales[“perro”] = ‘ladra’;
```

```
  animales[“vaca”] = ‘muge’;

  print(animales.keys);
  print(animales.values);
  for (var element in animales.entries) {
    print(‘${element.key}: ${element.value}’);
  }
}
```

Otras propiedades de los maps

- key
- values
- length
- isEmpty
- isNotEmpty

Salida:

```
(perro, vaca, leon)
(ladra, muge, ruge)
perro: ladra
vaca: muge
leon: ruge
```

Ejemplo: Métodos addAll, clear, remove, removeWhere

```
void main() {
  var animales = Map();

  animales[“leon”] = ‘ruge’;
  animales[“perro”] = ‘ladra’;
  animales[“vaca”] = ‘muge’;

  animales.addAll({“ave”: “canta”}); //agrega los elementos al
Map
  print(animales);
  print(“---------”);
  animales.clear(); //elimina todos los elementos del Map
  print(animales);
  print(“---------”);
  animales[“leon”] = ‘ruge’;
  animales[“perro”] = ‘ladra’;
  animales[“vaca”] = ‘muge’;
```

```
  print(animales);
  print("---------");
  animales.remove("perro");
  print(animales);
  print("---------");
  animales.removeWhere((key, value) => value == "muge");
  print(animales);
  print("---------");
  animales["leon"] = 'ruge';
  animales["perro"] = 'ladra';
  animales["vaca"] = 'muge';
  animales.removeWhere((key, value) => key == "leon" || value ==
"muge");
  print(animales);
  print("---------");
}
```

Salida:

```
{leon: ruge, perro: ladra, vaca: muge, ave: canta}
---------
{}
---------
{leon: ruge, perro: ladra, vaca: muge}
---------
{leon: ruge, vaca: muge}
---------
{leon: ruge}
---------
{perro: ladra}
---------
```

Enums

- Son tipo constantes.

Ejemplo: Enum del clima

```
enum Clima { climaSoleado, climaLluvioso, climaNublado }

void main() {
  Clima climaLunes = Clima.climaSoleado;
  print(climaLunes);
}
```

Salida:

```
Clima.climaSoleado
```

Ejemplo: Listar todos los valores de un Enum (colores)

```
enum Colores { rojo, amarillo, azul, verde, blanco }

void main() {
  print(Colores.values);
  print("-----------");
  for (var i = 0; i < Colores.values.length; i++) {
    print(Colores.values[i]);
  }
  print("-----------");
  for (var color in Colores.values) {
    print(color);
  }
  print("-----------");
  print(Colores.values[4]);
}
```

Salida:

```
[Colores.rojo, Colores.amarillo, Colores.azul, Colores.verde,
Colores.blanco]
-----------
Colores.rojo
Colores.amarillo
Colores.azul
Colores.verde
Colores.blanco
-----------
Colores.rojo
Colores.amarillo
Colores.azul
Colores.verde
Colores.blanco
-----------
Colores.blanco
```

Ejemplo: Enum para los días de la semana

```
enum DiaSemana {
  lunes,
  martes,
  miercoles,
  jueves,
  viernes,
  sabado,
  domingo
}

void main() {
  // Uso básico del enum
  DiaSemana hoy = DiaSemana.lunes;
  print('Hoy es: $hoy'); // Output: Hoy es: DiaSemana.lunes

  // Obtener el nombre del día
  print('Día: ${hoy.name}'); // Output: Día: lunes

  // Iterar sobre todos los días
  print('\nTodos los días de la semana:');
  for (var dia in DiaSemana.values) {
    print(dia.name);
  }

  // Usar en un switch
  String mensaje = switch (hoy) {
    DiaSemana.lunes => '¡Inicio de semana!',
    DiaSemana.viernes => '¡Ya casi es fin de semana!',
    DiaSemana.sabado || DiaSemana.domingo => '¡Fin de semana!',
    _ => 'Día regular de la semana'
  };
  print('\n$mensaje');

  // Comparación
  if (hoy == DiaSemana.lunes) {
    print('\nEs lunes, a trabajar!');
  }
}
```

Salida:

```
Hoy es: DiaSemana.lunes
Día: lunes

Todos los días de la semana:
lunes
martes
miercoles
jueves
viernes
sabado
domingo

Día regular de la semana
Es lunes, a trabajar!
```

Ejercicio: Crear un enum para los meses del año

```
enum Mes {
  enero,
  febrero,
  marzo,
  abril,
  mayo,
  junio,
  julio,
  agosto,
  septiembre,
  octubre,
  noviembre,
  diciembre
}

void main() {
  // Usar el enum
  Mes mesActual = Mes.octubre;
  print('Mes actual: $mesActual'); // Imprime: Mes actual: Mes.
octubre

  // Obtener el índice (empezando desde 0)
  print('Índice: ${mesActual.index}'); // Imprime: Índice: 9
```

```
  // Obtener el nombre
  print('Nombre: ${mesActual.name}'); // Imprime: Nombre: octubre

  // Iterar sobre todos los meses
  print('\nTodos los meses:');
  for (var mes in Mes.values) {
    print('${mes.index + 1}. ${mes.name}');
  }

  // Switch con enum
  String obtenerEstacion(Mes mes) {
    switch (mes) {
      case Mes.diciembre:
      case Mes.enero:
      case Mes.febrero:
        return 'Invierno';
      case Mes.marzo:
      case Mes.abril:
      case Mes.mayo:
        return 'Primavera';
      case Mes.junio:
      case Mes.julio:
      case Mes.agosto:
        return 'Verano';
      case Mes.septiembre:
      case Mes.octubre:
      case Mes.noviembre:
        return 'Otoño';
    }
  }

  print('\nEstación de ${mesActual.name}: ${obtenerEstacion(mes-
Actual)}');
}
```

Características de los enums en Dart

- index: devuelve la posición (empezando desde 0).
- name: devuelve el nombre como String.
- values: lista con todos los valores del enum.
- Son muy útiles para switches, ya que Dart verifica que cubras todos los casos.

Salida:

```
Mes actual: Mes.octubre
Índice: 9
Nombre: octubre

Todos los meses:
1. enero
2. febrero
3. marzo
4. abril
5. mayo
6. junio
7. julio
8. agosto
9. septiembre
10. octubre
11. noviembre
12. diciembre
```

Estación de octubre: Otoño

Esta salida muestra:

1. El mes actual con su representación completa.
2. Su índice (9, porque empieza desde 0).
3. Sólo el nombre del mes.
4. Una lista de todos los meses numerados del 1 al 12.
5. La estación correspondiente al mes de octubre.

Colas (queues)

Ejemplo: Crear una cola con nombres

```
import “dart:collection”;

void main() {
  Queue colaNombres = Queue<String>();
  //Queue<String> colaNombres = Queue<String>();

  colaNombres.add(“Hugo”);
  colaNombres.add(“Paco”);
  colaNombres.add(“Luis”);
```

```
  print(colaNombres);
}
Salida:
{Hugo, Paco, Luis}
Ejemplo: Agregar varios elementos al mismo tiempo a la cola.
void main() {
  Queue colaNombres = Queue();

  colaNombres.addAll([“Hugo”, “Paco”, “Luis”]);

  print(colaNombres);
}
```

Salida:

```
{Hugo, Paco, Luis}
```

Ejemplo: Agregar elementos al principio y al final de la cola

```
import “dart:collection”;

void main() {
  Queue<String> cola = Queue<String>();

  cola.add(“Paco”);
  cola.add(“Luis”);

  print(cola);

  cola.addFirst(“Hugo”);
  print(cola);

  cola.addLast(“Maria”);

  print(cola);
}
```

Salida:

```
{Paco, Luis}
{Hugo, Paco, Luis}
{Hugo, Paco, Luis, Maria}
```

Ejemplo: Eliminar el primero y el último elemento de la cola

```
import “dart:collection”;

void main() {
  Queue colaNombres = Queue();

  colaNombres.addAll([“Hugo”, “Paco”, “Luis”]);

  colaNombres.removeFirst();
  print(colaNombres);

  colaNombres.removeLast();
  print(colaNombres);
}
```

Salida:

```
{Paco, Luis}
{Paco}
```

Ejemplo: Eliminar determinado elemento de la cola

```
import “dart:collection”;

void main() {
  Queue colaNombres = Queue();

  colaNombres.addAll([“Hugo”, “Paco”, “Luis”, “Maria”, “Paco”,
“Romina”]);

  colaNombres.remove(“Paco”);
  print(colaNombres);

}
```

Salida:

```
{Hugo, Luis, Maria, Paco, Romina}
```

Ejemplo: Eliminar determinados elementos de la cola (repetidos) con función anónima

```
import “dart:collection”;

void main() {
  Queue colaNombres = Queue();
```

```
  colaNombres.addAll([“Hugo”, “Paco”, “Luis”, “Maria”, “Paco”,
“Romina”]);
  print(colaNombres);

  colaNombres.removeWhere((element) {
    return element == “Paco”;
  });
  print(colaNombres);
}
```

Salida:

```
{Hugo, Paco, Luis, Maria, Paco, Romina}
{Hugo, Luis, Maria, Romina}
```

Ejemplo: Eliminar determinados elementos de la cola (repetidos) con fat arrow function

```
import “dart:collection”;

void main() {
  Queue colaNombres = Queue();

  colaNombres.addAll([“Hugo”, “Paco”, “Luis”, “Maria”, “Paco”,
“Romina”]);
  print(colaNombres);

  colaNombres.removeWhere((element) => element == “Paco”);
  print(colaNombres);
}
```

Salida:

```
{Hugo, Paco, Luis, Maria, Paco, Romina}
{Hugo, Luis, Maria, Romina}
```

Ejemplo: Obtener un elemento por su posición en la cola

```
import “dart:collection”;

void main() {
  Queue colaNombres = Queue();

  colaNombres.addAll([“Hugo”, “Paco”, “Luis”, “Maria”, “Paco”,
```

```
“Romina”]);

  print(colaNombres);
  print(colaNombres.elementAt(2));
  print(“----”);
  colaNombres.forEach((element) {
    print(element);
  });
}
```

Salida:

```
{Hugo, Paco, Luis, Maria, Paco, Romina}
Luis
----
Hugo
Paco
Luis
Maria
Paco
Romina
```

Listas enlazadas (linked lists)

```
import “dart:collection”;

class Entry<T> extends LinkedListEntry<Entry<T>> {
  T value;
  Entry(this.value);
}

void main() {
  LinkedList listaEnlazada = LinkedList<Entry<int>>();
  listaEnlazada.add(Entry(1));
  listaEnlazada.add(Entry(2));
  listaEnlazada.add(Entry(3));

  print(listaEnlazada.first.next.value);
  print(listaEnlazada.elementAt(1).previous.value);
}
```

Salida:

```
2
1
```

Capítulo 6
Funciones

En este capítulo adentrémonos en manejo de las funciones en Dart. Las funciones son elementos fundamentales en la programación, ya que nos permiten encapsular lógica, reutilizar código y estructurar nuestros programas de manera eficiente y organizada.

Exploraremos distintos tipos de funciones, desde aquellas que realizan acciones específicas hasta aquellas que retornan valores. Conoceremos cómo trabajar con parámetros, tanto obligatorios como opcionales, así como a utilizar funciones anónimas y closures, que amplían nuestras posibilidades de desarrollo.

Entender a fondo el funcionamiento y la utilidad de las funciones es esencial para cualquier programador, ya que nos brindan la capacidad de modularizar nuestro código y hacerlo más legible, mantenible y eficiente. A continuación, te comentamos algunas funciones de Dart:

- Es un segmento de código que puede ser reutilizable las veces necesarias
- Pueden llevar o no argumentos/parámetros
- Pueden o no retornar valores
- Se recomienda que siempre se les indique el tipo de dato de la función (tipo de dato de retorno)

Funciones tipo procedimiento

- Las funciones tipo procedimiento no retornan ningún valor.
- Ejecutan sólo el código que está dentro del bloque de la función.
- El tipo de dato de la función es void.
- Se puede omitir el tipo de dato, pero es recomendable siempre tipar las funciones.

Ejemplo: Función por invocación/llamada sin argumentos/parámetros que imprime un mensaje

```
void main() {
  saludo();
}

void saludo() {
  print(“Hola Mundo”);
}
```

Ejemplo: Función sin argumentos/parámetros por invocación/llamada que calcula el perímetro de un cuadrado

```
void main() {
  perimetroCuadrado();
}

void perimetroCuadrado() {
  int lado = 5;
  int perimetro = lado * 4;
  print(“El perimetro es $perimetro”);
}
```

Ejemplo: Función con argumentos/parámetros por invocación/llamada que calcula el perímetro de un cuadrado

```
void main() {
  perimetroCuadrado(5);
}

void perimetroCuadrado(int lado) {
  int perimetro = lado * 4;
  print(“El perimetro es $perimetro”);
}
```

Salida:

```
El perimetro es 20
```

Funciones con retorno de valores

- Se recomienda especificar el tipo de dato que va a retornar la función.
- Debe existir una sentencia return con el valor a retornar.
- Pueden o no llevar argumentos/parámetros.

Ejemplo: Función que calcula el perímetro de un cuadrado por invocación o llamada interpolada dentro de la cadena

- La función print recibe el valor y lo imprime en la cadena.
- No se debe invocar la función directamente como si fuera procedimiento, pues el valor se perdería.

```
void main() {
  print(“El perímetro es ${perimetroCuadrado()}”);
  perime
```

```
}

int perimetroCuadrado() {
  int lado = 5;
  return lado * 4;
}
```

Salida:

```
El perímetro es 20
```

Ejemplo: Función sin argumentos/parámetros que calcula el perímetro de un cuadrado por asignación de la función, interpolando el resultado dentro de la cadena

```
void main() {
  var perimetro = perimetroCuadrado();
  print(“El perímetro es $perimetro”);
}

int perimetroCuadrado() {
  int lado = 5;
  return lado * 4;
}
```

Salida:

```
El perímetro es 20
```

Ejemplo: Función sin argumentos/parámetros que calcula el perímetro de un cuadrado por invocación/llamada de la función

```
void main() {
  print(“El perímetro es ${perimetroCuadrado(5)}”);
}

int perimetroCuadrado(int lado) {
  return lado * 4;
}
```

Ejemplo: Calculadora simple

```
void main(List<String> args) {
  List<dynamic> n = [20.5, 3.0];
  var res;
  var op = Operacion.Sumar;
  switch (op) {
```

```
    case Operacion.Sumar:
      res = n[0] + n[1];
      break;
    case Operacion.Restar:
      res = n[0] - n[1];
      break;
    case Operacion.Multiplicar:
      res = n[0] * n[1];
      break;
    case Operacion.Dividir:
      res = n[0] / n[1];
      break;

    default:
      print(“Operación desconocida”);
  }
  print(“Resultado: $res”);
}
```

Salida:

```
Resultado: 23.5
enum Operacion { Sumar, Restar, Multiplicar, Dividir }

void main(List<String> args) {
  List<dynamic> n = [20.5, 3.0];
  var res = calcular(n[0], n[1], Operacion.Sumar);
  print(“Resultado: $res”);
}

double calcular(double a, double b, Operacion op) {
  var res;
  switch (op) {
    case Operacion.Sumar:
      res = a + b;
      break;
    case Operacion.Restar:
      res = a - b;
      break;
    case Operacion.Multiplicar:
      res = a * b;
      break;
```

```
    case Operacion.Dividir:
      res = a / b;
      break;

    default:
      print("Operación desconocida");
  }
  return res;
}
```

Salida:

```
Resultado: 23.5
```

Funciones flecha/cortas (fat arrow)

Ejemplo: Calcular el área de un cuadrado

```
void main() {
  var res = areaCuadrado(5);
  print("El área es ${res}");
}

int areaCuadrado(int lado) => lado * lado;
```

Salida:

```
El área es 25
```

Ejemplo: Obtener el mayor y el menor de dos números enteros

```
void main() {
  int a = 10;
  int b = 15;
  print("El mayor entre $a y $b es ${mayor(a, b)}");
  print("El menor entre $a y $b es ${menor(a, b)}");
}

int mayor(a, b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
```

```
}

int menor(a, b) => (a < b) ? a : b;
```

Parámetros obligatorios, opcionales y nombrados

Ejemplo: Parámetros obligatorios, función que despliega el nombre y la edad de una persona

```
void main() {
  paramObligatorios('Alex', 50);
}

void paramObligatorios(String nombre, int edad) {
  print("Nombre: $nombre\nEdad:$edad");
}
```

Salida

```
Nombre: Alex
Edad:50
```

Ejemplo: Parámetros opcionales, función que despliega el nombre y la edad de una persona

```
void main() {
  paramOpcional1("Walter");
  paramOpcional1("Alex", 51);
  paramOpcional2();
  paramOpcional2("Alex");
  paramOpcional2("Alex", 51);
}

void paramOpcional1(String nombre, [int edad = 0]) {
  print("Nombre: $nombre\nEdad: $edad");
}

void paramOpcional2([String nombre = "", int edad = 0]) {
  print("Nombre: $nombre\nEdad: $edad");
}
```

Salida:

```
Nombre: Walter
```

```
Edad: 0
Nombre: Alex
Edad: 51
Nombre:
Edad: 0
Nombre: Alex
Edad: 0
Nombre: Alex
Edad: 51
```

Ejemplo: Parámetros nombrados, función que despliega el nombre y la edad de una persona

```
void main() {
  paramNombrados(nombre: “Walter”);
  paramNombrados(edad: 51);
  paramNombrados(edad: 51, nombre: “Alex”);
  paramNombrados(nombre: “Alex”, edad: 51);
}

void paramNombrados({String nombre = “”, int edad = 0}) {
  print(“Nombre: $nombre\nEdad: $edad”);
}
```

Salida:

```
Nombre: Walter
Edad: 0
Nombre:
Edad: 51
Nombre: Alex
Edad: 51
Nombre: Alex
Edad: 51
```

Ejemplo: Simulación de descarga de un archivo mediante una función que recibe como argumento el nombre del archivo y un valor booleano para abrir el archivo una vez descargado

Versión 1:

```
void main() {
  download(“carta.txt”);
  print(“------”);
  download(“cancion.mp4”, true);
```

```
}

void download(String fileName, [bool open = false]) {
  print(“downloading file $fileName...”);
  if (open) print(“Opening file $fileName...”);
}
```

Salida:

```
downloading file carta.txt...
------
downloading file cancion.mp4...
Opening file cancion.mp4...
```

Versión 2:

```
void main() {
  download(“carta.txt”);
  print(“------”);
  download(“cancion.mp4”, open: true);
}

void download(String fileName, {bool open = false}) {
  print(“downloading file $fileName...”);
  if (open) print(“Opening file $fileName...”);
}
```

Salida:

```
downloading file carta.txt...
------
downloading file cancion.mp4...
Opening file cancion.mp4...
```

Versión 3:

```
void main() {
  download(fileName: “carta.txt”);
  print(“------”);
  download(
    fileName: “cancion.mp4”,
    open: true,
  );
}

void download({String fileName = “”, bool open = false}) {
```

```
  print(“downloading file $fileName...”);
  if (open) print(“Opening file $fileName...”);
}
```

Salida:

```
downloading file carta.txt...
------
downloading file cancion.mp4...
Opening file cancion.mp4...
```

Funciones anónimas

- No tienen nombre.
- Sirven para ejecutar un segmento de código una sola vez.
- No se pueden reutilizar.

Ejemplo: Imprimir los elementos de una lista de personas mediante una función anónima

Versión 1:

```
void main() {
  List personas = [“Hugo”, “Paco”, “Luis”];

  personas.forEach((persona) {
    print(persona);
  });
}
```

Salida:

```
Hugo
Paco
Luis
```

Versión 2:

```
void main() {
  List<String> personas = <String>[“Hugo”, “Paco”, “Luis”];

  personas.forEach((persona) => print(persona));
}
```

Salida:

```
Hugo
Paco
Luis
```

Versión 3:

```
void main() {
  List<String> personas = <String>[“Hugo”, “Paco”, “Luis”];

  personas.forEach(print);
}
```

Salida:

```
Hugo
Paco
Luis
```

Versión 4:

```
void main() {
  List<String> personas = <String>[“Hugo”, “Paco”, “Luis”];

  personas.forEach((persona) => desplegar(persona));
}

void desplegar(String persona) {
  print(persona);
}
```

Salida:

```
Hugo
Paco
Luis
```

Versión 5:

```
void main() {
  List<String> personas = <String>[“Hugo”, “Paco”, “Luis”];

  personas.forEach(desplegar);
}

void desplegar(String persona) {
  print(persona);
}
```

Salida:

```
Hugo
Paco
Luis
```

Salida:

```
Hugo
Paco
Luis
```

Asignación de funciones

Ejemplo: Calcular el área de un cuadrado

```
void main() {
  var area1 = areaCuadrado;
  print(area1.runtimeType);

  var area2 = areaCuadrado(2);

  print(area1(area2));
  print(area2);
  //print(area2(3));

  var area3 = areaCuadrado;
  print(area3(3));
}

int areaCuadrado(int lado) => lado * lado;
```

Salida:

```
16
4
9
```

Funciones recursivas

- Capacidad que tiene una función de llamarse a sí misma.

Ejemplo: Obtener el factorial de un número

Versión 1:

```
void main() {
  int n = 5;

  print("$n! = ${factorialSinRecursion(n)}");
  print("$n! = ${factorialRecursivo(n)}");
}
```

```
int factorialSinRecursion(int n) {
  int fact = 1;
  for (var i = 1; i <= n; i++) {
    fact *= i;
  }
  return fact;
}

int factorialRecursivo(int n) {
  if (n <= 0) {
    return 1;
  } else {
    int fact = (n * factorialRecursivo(n - 1));
    return fact;
  }
}
```

Salida:

```
5! = 120
5! = 120
```

Closures

- Es una función retornada desde otra función, la cual puede acceder a las variables de la función superior.

Ejemplo:

```
void main() {
  print(cuad());
}

cuad() {
  return () {
    return 2 * 2;
  };
}
```

Salida:

```
Closure: () => int
```

Ejemplo:

```
void main() {
  print(cuad()());
}

cuad() {
  return () {
    return 2 * 2;
  };
}
```

Salida:

```
4
```

Ejemplo:

```
void main() {
  print(cuad()());
}

Function cuad() {
  return () {
    return 2 * 2;
  };
}
```

Salida:

```
4
```

Ejemplo:

```
void main() {
  print(cuad(2)());
}

Function cuad(int n) {
  int result = n * n;
  return () {
    return result;
  };
}
```

Salida:

```
4
```

Ejemplo:

```
void main() {
  print(mul(2)(3));
}

Function mul(int n) {
  var nn = n;
  return (int m) {
    return nn * m;
  };
}
```

Salida:

```
6
```

Ejemplo:

```
void main() {
  var op = mul(2);
  print(op(3));
}

Function mul(int n) {
  var nn = n;
  return (int m) {
    return nn * m;
  };
}
```

Salida:

```
6
```

Capítulo 7
Manejo de errores y excepciones

Toca el turno de tratar el tema del manejo de errores y excepciones en Dart, exploraremos varios aspectos clave que permiten a los desarrolladores gestionar situaciones inesperadas durante la ejecución de un programa. Desde el manejo de errores específicos, como la división entera por cero, hasta la implementación de excepciones personalizadas, conoceremos las herramientas y técnicas fundamentales para garantizar una respuesta controlada y adecuada ante distintos escenarios. Analizaremos ejemplos prácticos de uso de bloques try/catch, try/on, try/on/catch, y finally, brindando una comprensión completa sobre cómo abordar y mitigar errores en nuestras aplicaciones en Dart. A continuación, te mostramos algunas de sus características:

- Un Error es una falla en el programa.
- Una Excepción es un error controlado.

Ejemplo: Error de división entera por 0 Bloque try/catch

```
void main(List<String> args) {
  int a = 10;
  int b = 0;

  int res;
  res = a ~/ b;
  print(res);

}
```

Salida:

```
Unhandled exception:
IntegerDivisionByZeroException
#0      int.~/ (dart:core-patch/integers.dart:30:7)
#1      main (file:///D:/wkspaceLearnDart/mi_app/bin/mi_app.
dart:17:11)
#2      _delayEntrypointInvocation.<anonymous closure> (dart:iso-
late-patch/isolate_patch.dart:295:32)
#3      _RawReceivePortImpl._handleMessage (dart:isolate-patch/
isolate_patch.dart:192:12)
```

Ejemplo: Error de división entera por 0 con bloque try/catch

```
void main(List<String> args) {
  int a = 10;
  int b = 0;

  int res;

  try {
    res = a ~/ b;
    print(res);
  } catch (error) {
    print(“Error división por 0: ${error.toString()}”);
  }
}
```

Salida:

```
Error división por 0: IntegerDivisionByZeroException
```

Ejemplo: Bloque try/on

Versión 1:

```
void main(List<String> args) {
  int a = 10;
  int b = 0;

  int res;

  try {
    res = a ~/ b;
    print(res);
  } on IntegerDivisionByZeroException {
    print(“Error división por 0”);
  }
}
```

Salida:

```
Error división por 0
```

Versión 2:

```
void main(List<String> args) {
  int a = 10;
  int b = 0;
```

```
  int res;

  try {
    res = a ~/ b;
    print(res);
  } on UnsupportedError {
    print("Error división por 0");
  }
}
```

Salida:

```
Error división por 0
```

Ejemplo: Bloque try/on/catch

- En caso de que exista otra excepción distinta a la establecida en **on UnsupportedError** entrará al catch.

```
void main(List<String> args) {
  int a = 10;
  int b = 0;

  int res;

  try {
    res = a ~/ b;
    print(res);
  } on UnsupportedError {
    print("Error división por 0");
  } catch (error) {
    print("Error: ${error.toString()}");
  }
}
```

Salida:

```
Error división por 0
```

Ejemplo: Bloque finally

- Independientemente de las excepciones, el código dentro del bloque finally siempre se va a ejecutar.
- Puede ayudar a cerrar archivos, por ejemplo, si se está haciendo uso de ellos, esto para que no queden abiertos y se puedan dañar.

Versión 1:

```
void main(List<String> args) {
  int a = 10;
  int b = 0;

  int res;

  try {
    res = a ~/ b;
    print(res);
  } on UnsupportedError {
    print(“Error división por 0”);
  } catch (error) {
    print(“Error: ${error.toString()}”);
  } finally {
    print(“Fin del programa...”);
  }
}
```

Salida:

```
Error división por 0
Fin del programa...
```

Versión 2:

```
void main(List<String> args) {
  int a = 10;
  int b = 3;

  int res;

  try {
    res = a ~/ b;
    print(res);
  } on UnsupportedError {
    print(“Error división por 0”);
  } catch (error) {
    print(“Error: ${error.toString()}”);
  } finally {
    print(“Fin del programa...”);
  }
}
```

Salida:

```
3
Fin del programa...
```

Ejemplo: Excepciones personalizadas

```
void main(List<String> args) {
  int a = 10;
  int b = 0;

  int res;

  try {
    res = division(a, b);
    print(res);
  } on DivisionPorCeroException catch (error) {
    print("Error: $error");
  } catch (error) {
    print("Error: ${error.toString()}");
  } finally {
    print("Fin del programa...");
  }
}

class DivisionPorCeroException implements Exception {
  @override
  String toString() {
    return "No se puede realizar una división por cero..";
  }
}

int division(int a, int b) {
  if (b == 0) {
    throw DivisionPorCeroException();
  } else {
    return a ~/ b;
  }
}
```

Salida:

```
Error: No se puede realizar una división por cero..
Fin del programa...
```

Conclusiones de la parte I

En esta parte dedicada a la programación estructurada con Dart, hemos explorado exhaustivamente los fundamentos y elementos esenciales de este lenguaje de programación versátil y moderno. Comenzamos con una introducción detallada, abordando la historia de Dart y sus características principales, así como las posibilidades de desarrollo que ofrece.

A lo largo de esta parte, hemos examinado diversos temas cruciales para un programador en Dart. Desde la manipulación de variables y tipos de datos, incluyendo el manejo de cadenas, numéricos y lógicos, hasta la implementación de estructuras condicionales y bucles, hemos cubierto una amplia gama de conceptos clave. También hemos explorado el uso de colecciones como listas, sets, maps y colas, así como el manejo de funciones, parámetros y excepciones.

Es importante resaltar la importancia de comprender cómo trabajar con datos y estructuras en Dart, así como la habilidad para crear funciones y utilizarlas de manera efectiva en la construcción de programas. Asimismo, hemos abordado el crucial tema del manejo de errores y excepciones, destacando la utilización de bloques try/catch, try/on, try/on/catch, finally, y la creación de excepciones personalizadas para garantizar un desarrollo robusto y confiable.

Al abordar la diversidad de conceptos y técnicas, desde el uso de variables y estructuras de control hasta la implementación de funciones recursivas y programación asíncrona, consideramos que hemos proporcionado un panorama completo para que los desarrolladores se adentren en la programación estructurada con Dart y amplíen sus habilidades en este versátil lenguaje. Esta base sólida sienta las bases para explorar en profundidad la Programación Orientada a Objetos con Dart, tema central de la siguiente parte.

Parte II
Programación orientada a objetos con Dart

En esta segunda parte titulada “Programación orientada a objetos con Dart”, daremos un paso fundamental hacia el dominio de la programación orientada a objetos (POO) en el contexto del lenguaje de programación Dart. La programación orientada a objetos es una metodología esencial que permite organizar y estructurar el código en términos de objetos, clases y relaciones entre ellos, facilitando la reutilización, extensión y mantenimiento del código.

Exploraremos en profundidad el concepto de clases, que constituyen la piedra angular de la POO en Dart. Veremos cómo crear clases, instanciarlas y trabajar con propiedades y métodos. Adentrándonos más, examinaremos los diferentes tipos de constructores, incluyendo argumentos opcionales y constructores nombrados, que nos brindan flexibilidad en la inicialización de objetos.

Posteriormente, nos sumergiremos en el mundo de la herencia, un concepto esencial en la POO, que nos permite crear nuevas clases basadas en clases existentes, facilitando la reutilización y extensión del código. Además, exploraremos otras características avanzadas como clases abstractas, mixins, interfaces y genéricos, que enriquecen nuestras capacidades de diseño y programación.

Finalmente, abordaremos la programación asíncrona en Dart, éste es un tema crucial en el entorno moderno de la programación, que nos permite ejecutar tareas de manera eficiente y sin bloqueos, mejorando así la eficacia y rendimiento de nuestras aplicaciones.

Con el contenido de esta sección, sentaremos las bases para una comprensión profunda y sólida de la programación orientada a objetos en Dart, proporcionando las herramientas esenciales para abordar proyectos complejos y construir aplicaciones robustas y flexibles.

Capítulo 8
Clases

En este capítulo dedicado a las clases en Dart, exploraremos un concepto fundamental en la programación orientada a objetos. Las clases proporcionan una estructura organizada y reutilizable para definir objetos y sus propiedades, así como los comportamientos asociados a ellos. A lo largo de esta sección, examinaremos ejemplos prácticos que abarcan desde la creación de una clase básica hasta la instancia de objetos utilizando propiedades y métodos. Veremos cómo diseñar y utilizar clases en Dart para construir aplicaciones eficientes y flexibles. Comencemos adentrándonos en el mundo de las clases y su aplicación en este potente lenguaje de programación.

Ejemplo: Creación de una clase

```
void main(List<String> args) {
  print("Clase: ${Persona().runtimeType}");
}

class Persona {}
```

Salida:

```
Clase: Persona
```

Ejemplo: Creación de una instancia de clase

```
void main(List<String> args) {
  Persona persona = Persona();
  print(persona);
  print("Clase: ${persona.runtimeType}");
}

class Persona {}
```

Salida:

```
Instance of 'Persona'
Clase: Persona
```

Ejemplo: Creación e instanciación de la clase Persona con propiedades

```
void main(List<String> args) {
  Persona persona = Persona();
  print(persona.nombre);
  print(persona.edad);
}

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

Salida:

```
null
null
```

Ejemplo: Uso de las propiedades de instancia

```
void main(List<String> args) {
  Persona persona = Persona();
  print(persona.nombre);
  print(persona.edad);
  persona.nombre = “Alex”;
  persona.edad = 51;
  print(persona.nombre);
  print(persona.edad);
}

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

Salida:

```
null
null
Alex
51
```

Ejemplo: Creación de métodos en la clase

```
void main(List<String> args) {
  Persona persona = Persona();
```

```
  persona.mostrarDatosPersona();
}

class Persona {
  //Propiedades
  String? nombre;
  int? edad;

  //Métodos
  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: $edad”);
  }
}
```

Salida:

```
Nombre: null
Edad: null
```

Ejemplo: Instancia de la clase Persona usando propiedades de instancia

```
void main(List<String> args) {
  Persona persona = Persona();

  persona.nombre = “Hugo”;
  persona.edad = 10;
  print(“Nombre: ${persona.nombre}”);
  print(“Edad: ${persona.edad}”);
}

class Persona {
  //Propiedades
  String? nombre;
  int? edad;

  //Métodos
  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: $edad”);
  }
}
```

Salida:

```
Nombre: Hugo
Edad: 10
```

Ejemplo: Instancia de la clase Persona usando propiedades y métodos de instancia

```
void main(List<String> args) {
  Persona persona = Persona();

  persona.nombre = “Hugo”;
  persona.edad = 10;
  persona.mostrarDatosPersona();
}

class Persona {
  //Propiedades
  String? nombre;
  int? edad;

  //Métodos
  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: $edad”);
  }
}
```

Salida:

```
Nombre: Hugo
Edad: 10
```

Capítulo 9
Constructores

En este capítulo dedicada a los constructores en Dart, nos adentraremos en un aspecto crucial de la programación orientada a objetos. Los constructores son métodos especiales que permiten inicializar objetos de una clase y configurar su estado inicial. Exploraremos varios ejemplos para comprender cómo diseñar y utilizar diferentes tipos de constructores en Dart. Desde constructores básicos hasta aquellos con argumentos opcionales, sobrecarga de constructores y acceso a propiedades, analizaremos en detalle cada caso. Comprender la función de los constructores en la creación y configuración de objetos es fundamental para un desarrollo efectivo en Dart, así que exploremos este concepto esencial en la programación orientada a objetos.

Ejemplo: Constructor de la clase Persona

```
void main(List<String> args) {
  Persona persona = Persona(“Hugo”, 10);

  print(“Nombre: ${persona.nombre}”);
  print(“Edad: ${persona.edad}”);
}

class Persona {
  //Propiedades
  String? nombre;
  int? edad;

  //Constructor
  Persona(String nom, int ed) {
    nombre = nom;
    edad = ed;
  }

  //Métodos
  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: $edad”);
  }
}
```

Salida:

```
Nombre: Hugo
Edad: 10
```

Ejemplo: Constructor de la clase Persona con el uso del operador this

Versión 1:

```
void main(List<String> args) {
  Persona persona = Persona(“Hugo”, 10);

  print(“Nombre: ${persona.nombre}”);
  print(“Edad: ${persona.edad}”);
}

class Persona {
  String? nombre;
  int? edad;

  //Constructor
  Persona(String nombre, int edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: $edad”);
  }
}
```

Salida:

```
Nombre: Hugo
Edad: 10
```

Versión 2:

```
void main(List<String> args) {
  Persona persona = Persona(“Hugo”, 10);

  print(“Nombre: ${persona.nombre}”);
  print(“Edad: ${persona.edad}”);
}

class Persona {
```

```
  //Propiedades
  String? nombre;
  int? edad;

  Persona(this.nombre, this.edad);

  //Métodos
  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: $edad”);
  }
}
```

Salida:

```
Nombre: Hugo
Edad: 10
```

Ejemplo: Constructor de la clase persona con argumentos opcionales y por defecto

```
void main(List<String> args) {
  Persona persona = Persona(“Hugo”);

  print(“Nombre: ${persona.nombre}”);
  print(“Edad: ${persona.edad}”);
}

class Persona {
  //Propiedades
  String? nombre;
  int? edad;

  Persona(this.nombre, [this.edad = 18]);

  //Métodos
  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: $edad”);
  }
}
```

Salida:

```
Nombre: Hugo
Edad: 18
```

Ejemplo: Constructores nombrados (sobrecarga de constructores)

```
void main(List<String> args) {
  Persona persona1 = Persona.nombre(“Hugo”);
  Persona persona2 = Persona.edad(18);

  print(“Nombre: ${persona1.nombre}”);
  print(“Edad: ${persona1.edad}”);
  print(“Nombre: ${persona2.nombre}”);
  print(“Edad: ${persona2.edad}”);
}

class Persona {
  //Propiedades
  String? nombre;
  int? edad;

  Persona.nombre(this.nombre);
  Persona.edad(this.edad);

  //Métodos
  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: $edad”);
  }
}
```

Salida:

```
Nombre: Hugo
Edad: null
Nombre: null
Edad: 18
```

Ejemplo: Creación de clases en archivos independientes con el uso de import

1. Dentro de la carpeta lib creamos el archivo que va a contener la clase: Persona.dart
   Arhivo: lib/persona.dart

```
class Persona {
  //Propiedades
  String? nombre;
  int? edad;
```

```
  Persona.nombre(this.nombre);
  Persona.edad(this.edad);

  //Métodos
  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: $edad”);
  }
}
```

2. Eliminamos el código de la clase del programa principal (mi_app.dart)
3. Agregamos la instrucción *import* correspondiente

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Persona.dart’;

void main(List<String> args) {
  Persona persona1 = Persona.nombre(“Hugo”);
  Persona persona2 = Persona.edad(18);

  print(“Nombre: ${persona1.nombre}”);
  print(“Edad: ${persona1.edad}”);
  print(“Nombre: ${persona2.nombre}”);
  print(“Edad: ${persona2.edad}”);
}
```

Salida:

```
Nombre: Hugo
Edad: null
Nombre: null
Edad: 18
```

Ejemplo: Propiedades públicas y privadas

Archivo: lib/Persona.dart

```
class Persona {
  //Propiedades
  String? nombre;
  int? _edad;

  Persona.nombre(this.nombre);
  Persona.edad(this._edad);
```

```
  //Métodos
  void mostrarDatosPersona() {
    print("Nombre: $nombre");
    print("Edad: $_edad");
  }
}
```

Archivo: bin/mi_app.dart

```
import 'package:mi_app/Persona.dart';

void main(List<String> args) {
  Persona persona1 = Persona.nombre("Hugo");
  print("Nombre: ${persona1.nombre}");

  persona1.nombre = "Paco";
  print("Nombre: ${persona1.nombre}");

  persona1.mostrarDatosPersona();

  Persona persona2 = Persona.edad(18);
  /*print("Edad: ${persona2.edad}");
  persona2.edad = 25;

  persona2._edad = 30;
  print("Edad: ${persona2._edad}");
*/
  persona2.mostrarDatosPersona();
}
```

Salida:

```
Nombre: Hugo
Nombre: Paco
Nombre: Paco
Edad: null
Nombre: null
Edad: 18
```

Ejemplo: Métodos públicos y privados

Archivo: lib/Persona.dart

```
class Persona {
  //Propiedades
```

```
  String? nombre;
  int? aNacimiento;

  Persona(this.nombre, this.aNacimiento);

  //Métodos
  void mostrarDatosPersona() {
    print(“Nombre: $nombre”);
    print(“Edad: ${_calcularEdad(aNacimiento!)}”);
  }

  int _calcularEdad(int aNacimiento) => 2023 - aNacimiento;
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Persona.dart’;

void main(List<String> args) {
  Persona persona1 = Persona(“Hugo”, 2015);
  persona1.mostrarDatosPersona();
  //persona1._calcularEdad(persona1.aNacimiento);
}
```

Salida:

```
Nombre: Hugo
Edad: 8
```

Ejemplo: Getters y Setters

- Es recomendable dejar todas las propiedades de instancia privados.
- Los getters se crean con *get*.
- Los setters se crean con *set*.

Archivo: lib/Persona.dart

```
class Persona {
  //Propiedades
  String _nombre;
  int _aNacimiento;

  //Constructor
  Persona(this._nombre, this._aNacimiento);

  //Getters
```

```
  String get nombre => _nombre;
  int get aNacimiento => _aNacimiento;

  //Setters

  void set nombre(String nombre) => _nombre = nombre;
  void set aNacimiento(int aNacimiento) => _aNacimiento =
aNacimiento;

  //Métodos
  void mostrarDatosPersona() {
    print("Nombre: $_nombre");
    print("Edad: ${_calcularEdad(_aNacimiento)}");
  }

  int _calcularEdad(int aNacimiento) => 2023 - aNacimiento;
}
```

Archivo: bin/mi_app.dart

```
import 'package:mi_app/Persona.dart';

void main(List<String> args) {
  Persona persona = Persona("Hugo", 2015);
  persona.mostrarDatosPersona();
  print("-" * 10);

  persona.nombre = "Paco";
  persona.aNacimiento = 2020;
  persona.mostrarDatosPersona();
  print("-" * 10);

  persona.nombre = "Luis";
  persona.aNacimiento = 2015;
  print("Nombre: ${persona.nombre}");
  print("Edad: ${2023 - persona.aNacimiento}");
  print("-" * 10);

  persona.mostrarDatosPersona();
}
```

Salida:

```
Nombre: Hugo
Edad: 8
----------
Nombre: Paco
Edad: 3
----------
Nombre: Luis
Edad: 8
----------
Nombre: Luis
Edad: 8
```

Ejemplo: Clase Vehículo

- Debe tener dos atributos: color y velocidad.
- Crear el constructor.
- Crear los getters y setters.
- Crear un método que calcule la velocidad máxima (doble de la velocidad ingresada).
- Hacer el test de la aplicación con constructor, getters y setters.

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String _color;
  int _velocidad;

  Vehiculo(this._color, this._velocidad);

  String get color => _color;
  int get velocidad => _velocidad;

  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  int maxVelocidad(int velocidad) => velocidad * 2;
  //int maxVelocidad() => _velocidad * 2;
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Vehiculo.dart’;

void main(List<String> args) {
```

```
  Vehiculo miVehiculo = Vehiculo(“rojo”, 120);

  print(“Color: ${miVehiculo.color}”);
  print(“Velocidad: ${miVehiculo.velocidad}”);
  print(“Máxima velocidad: ${miVehiculo.maxVelocidad(miVehiculo.
velocidad)}”);

  print(“-” * 15);
  miVehiculo.color = “Azul”;
  miVehiculo.velocidad = 80;
  var maxVel = miVehiculo.maxVelocidad(miVehiculo.velocidad);
  print(“Color: ${miVehiculo.color}”);
  print(“Velocidad: ${miVehiculo.velocidad}”);
  print(“Máxima velocidad: $maxVel”);
}
```

Salida:

```
Color: rojo
Velocidad: 120
Máxima velocidad: 240
---------------
Color: Azul
Velocidad: 80
Máxima velocidad: 160
```

Capítulo 10
Propiedades y métodos de clase (static)

Este capítulo se dedica a las propiedades y métodos de clase en Dart, exploraremos un aspecto fundamental de la programación orientada a objetos. Las propiedades y métodos de clase son elementos que pertenecen a la clase en sí misma, en lugar de pertenecer a instancias individuales. A través de ejemplos concretos, examinaremos cómo utilizar propiedades para contar instancias creadas y cómo definir y llamar a métodos de clase. Comprender cómo trabajar con propiedades y métodos de clase es esencial para la organización eficaz de nuestro código y para llevar a cabo operaciones que son relevantes a la clase en su totalidad. Vamos a explorar este aspecto crucial de la programación orientada a objetos en Dart.

Ejemplo: Conteo de instancias de clases creadas (propiedades)

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  int _count = 0;
  static int countStatic = 0;
  Vehiculo() {
    _count++;
    countStatic++;
  }

  void showCounter() {
    print(“Propiedad de Instancia: $_count”);
    print(“Propiedad de Clase: $countStatic”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Vehiculo.dart’;

void main(List<String> args) {
  var miVehiculo1 = Vehiculo();
  miVehiculo1.showCounter();
  print(“-” * 10);
  var miVehiculo2 = Vehiculo();
```

```
  miVehiculo2.showCounter();
  print(“-” * 10);
  var miVehiculo3 = Vehiculo();
  miVehiculo3.showCounter();
}
```

Salida:

```
Propiedad de Instancia: 1
Propiedad de Clase: 1
----------
Propiedad de Instancia: 1
Propiedad de Clase: 2
----------
Propiedad de Instancia: 1
Propiedad de Clase: 3
```

Ejemplo: Métodos de Clase y métodos de Instancia

Archivo: lib/

```
class Vehiculo {
  int _count = 0;
  static int countStatic = 0;
  Vehiculo() {
    _count++;
    countStatic++;
  }

  static void showCounterStatic() {
    //print(“Contador de Instancia: $_count”);
    print(“Contador de Clase: $countStatic”);
  }

  void showCounter() {
    print(“Contador de Instancia: $_count”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Vehiculo.dart’;

void main(List<String> args) {
  var miVehiculo1 = Vehiculo();
```

```
  miVehiculo1.showCounter();
  Vehiculo.showCounterStatic();
  print(“-” * 10);

  var miVehiculo2 = Vehiculo();
  miVehiculo2.showCounter();
  Vehiculo.showCounterStatic();
  print(“-” * 10);

  var miVehiculo3 = Vehiculo();
  miVehiculo3.showCounter();
  Vehiculo.showCounterStatic();
  print(Vehiculo.countStatic);
}
```

Salida:

```
Contador de Instancia: 1
Contador de Clase: 1
----------
Contador de Instancia: 1
Contador de Clase: 2
----------
Contador de Instancia: 1
Contador de Clase: 3
3
```

Capítulo 11
Herencia

En este capítulo centrado en la herencia en Dart, nos adentraremos en un concepto fundamental de la programación orientada a objetos. La herencia es un mecanismo clave que permite la creación de nuevas clases basadas en clases ya existentes, lo que promueve la reutilización de código y la organización eficiente de nuestras aplicaciones. A través de diversos ejemplos, exploraremos cómo implementar herencia en Dart, cómo heredar propiedades y métodos de una clase padre, y cómo personalizar y extender estas propiedades y métodos en las clases hijas. La herencia es un pilar esencial en el diseño de software en Dart, así que exploremos este concepto crucial en la programación orientada a objetos.

Ejemplo: Clases Motocicleta y Carro que heredan de vehículo

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String? _color;
  int? _velocidad;

  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;
}
```

Archivo: lib/Carro.dart

```
import ‘Vehiculo.dart’;

class Carro extends Vehiculo {

}
```

Archivo: lib/Motocicleta.dart

```
import ‘package:mi_app/Vehiculo.dart’;

class Motocicleta extends Vehiculo {
```

```
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Carro.dart’;
import ‘package:mi_app/Motocicleta.dart’;
//import ‘package:mi_app/Vehiculo.dart’;

void main(List<String> args) {
  var carro1 = Carro();
  carro1.color = “rojo”;
  carro1.velocidad = 100;

  print(carro1.color);
  print(carro1.velocidad);
  print(carro1.maxVelocidad());
  print(“-” * 20);

  var moto1 = Motocicleta();
  moto1.color = “azul”;
  moto1.velocidad = 60;

  print(moto1.color);
  print(moto1.velocidad);
  print(moto1.maxVelocidad());
}
```

Salida:

```
rojo
100
200
--------------------
azul
60
120
```

Ejemplo: Clases Motocicleta y Carro que heredan de vehículo con su Constructor

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String? _color;
  int? _velocidad;
```

```
  //Vehiculo(this._color, this._velocidad);
  Vehiculo() {
    print("Costructor padre - Vehiculo");
  }
  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;
}
```

Archivo: lib/Carro.dart

```
import 'Vehiculo.dart';

class Carro extends Vehiculo {
  Carro() {
    print("Constructor hijo - Carro");
  }
}
```

Archivo: lib/Motocicleta.dart

```
import 'Vehiculo.dart';

class Motocicleta extends Vehiculo {
  Motocicleta() {
    print("Constructor hijo - Motocicleta");
  }
}
```

Archivo: bin/mi_app.dart

```
import 'package:mi_app/Carro.dart';
import 'package:mi_app/Motocicleta.dart';

void main(List<String> args) {
  var carro1 = Carro();
  carro1.color = "rojo";
  carro1.velocidad = 100;

  print(carro1.color);
  print(carro1.velocidad);
```

```
  print(carro1.maxVelocidad());
  print("-" * 20);

  var moto1 = Motocicleta();
  moto1.color = "azul";
  moto1.velocidad = 60;

  print(moto1.color);
  print(moto1.velocidad);
  print(moto1.maxVelocidad());
}
```

Salida:

```
Costructor padre - Vehiculo
Constructor hijo - Carro
rojo
100
200
--------------------
Costructor padre - Vehiculo
Constructor hijo - Motocicleta
azul
60
120
```

Ejemplo: Clases Motocicleta y Carro que heredan de vehículo con los constructores padres

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String? _color;
  int? _velocidad;

  Vehiculo(this._color, this._velocidad) {
    print("Constructor padre - Vehiculo");
  }
  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;
```

```
  void showVehicle() {
    print(“Color: $_color”);
    print(“Velocidad: $_velocidad”);
  }
}
```

Archivo: lib/Carro.dart

```
import ‘Vehiculo.dart’;

class Carro extends Vehiculo {
  Carro(super.color, super.velocidad) {
    print(“Constructor hijo -  Carro”);
  }
}
```

Archivo: lib/Motocicleta.dart

```
import ‘Vehiculo.dart’;

class Motocicleta extends Vehiculo {
  Motocicleta(super.color, super.velocidad) {
    print(“Constructor hijo - Motocicleta”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Carro.dart’;
import ‘package:mi_app/Motocicleta.dart’;

void main(List<String> args) {
  var carro1 = Carro(“rojo”, 100);
  carro1.showVehicle();
  print(carro1.maxVelocidad());
  print(“-” * 20);

  var moto1 = Motocicleta(“Azul”, 60);
  moto1.showVehicle();
  print(moto1.maxVelocidad());
}
```

Salida:

```
Constructor padre - Vehiculo
Constructor hijo -  Carro
Color: rojo
```

```
Velocidad: 100
200
--------------------
Cnostructor padre - Vehiculo
Constructor hijo - Motocicleta
Color: Azul
Velocidad: 60
120
```

Ejemplo: Clases Motocicleta y Carro que heredan de vehículo con propiedades en cada clase hija

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String? _color;
  int? _velocidad;

  //Vehiculo(this._color, this._velocidad);
  Vehiculo() {
    print("Costructor padre - Vehiculo");
  }
  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;
}
```

Archivo: lib/Carro.dart

```
import 'Vehiculo.dart';

class Carro extends Vehiculo {
  String _marca;

  Carro(this._marca) {
    print("Constructor hijo - Carro $_marca");
  }
}
```

Archivo: lib/Motocicleta.dart

```
import ‘Vehiculo.dart’;

class Motocicleta extends Vehiculo {
  String _modelo;
  Motocicleta(this._modelo) {
    print(“Constructor hijo - Motocicleta $_modelo”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Carro.dart’;
import ‘package:mi_app/Motocicleta.dart’;

void main(List<String> args) {
  var carro1 = Carro(“Ford”);
  carro1.color = “rojo”;
  carro1.velocidad = 100;

  print(carro1.color);
  print(carro1.velocidad);
  print(carro1.maxVelocidad());
  print(“-” * 20);

  var moto1 = Motocicleta(“REBEL 1100”);
  moto1.color = “azul”;
  moto1.velocidad = 60;

  print(moto1.color);
  print(moto1.velocidad);
  print(moto1.maxVelocidad());
}
```

Salida:

```
Costructor padre - Vehiculo
Constructor hijo - Carro Ford
rojo
100
200
--------------------
Costructor padre - Vehiculo
Constructor hijo - Motocicleta REBEL 1100
```

```
azul
60
120
```

Ejemplo: Clases Motocicleta y Carro que heredan de vehículo con el constructor padre y nuevas propiedades

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String? _color;
  int? _velocidad;

  Vehiculo(this._color, this._velocidad) {
    print(“Constructor padre - Vehiculo”);
  }
  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;

  void showVehicle() {
    print(“Color: $_color”);
    print(“Velocidad: $_velocidad”);
  }
}
```

Archivo: lib/Carro.dart

```
import ‘Vehiculo.dart’;

class Carro extends Vehiculo {
  String _marca;

  Carro(super.color, super.velocidad, this._marca) {
    print(“Constructor hijo -  Carro”);
  }
}
```

Archivo: lib/Motocicleta.dart

```
import ‘Vehiculo.dart’;

class Motocicleta extends Vehiculo {
  String _modelo;

  Motocicleta(super.color, super.velocidad, this._modelo) {
    print(“Constructor hijo - Motocicleta”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Carro.dart’;
import ‘package:mi_app/Motocicleta.dart’;

void main(List<String> args) {
  var carro1 = Carro(“rojo”, 100, “Ford”);
  carro1.showVehicle();
  print(carro1.maxVelocidad());
  print(“-” * 20);

  var moto1 = Motocicleta(“Azul”, 60, “T1000”);
  moto1.showVehicle();
  print(moto1.maxVelocidad());
}
```

Salida:

```
Constructor padre - Vehiculo
Constructor hijo -  Carro
Color: rojo
Velocidad: 100
200
--------------------
Constructor padre - Vehiculo
Constructor hijo - Motocicleta
Color: Azul
Velocidad: 60
120
```

Ejemplo: Clases Motocicleta y Carro que heredan de vehículo con sobreescritura del método padre

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String? _color;
  int? _velocidad;

  Vehiculo(this._color, this._velocidad) {
    print(“Constructor padre - Vehiculo”);
  }
  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;

  void showVehicle() {
    print(“Color: $_color”);
    print(“Velocidad: $_velocidad”);
  }
}
```

Archivo: lib/Carro.dart

```
import ‘Vehiculo.dart’;

class Carro extends Vehiculo {
  String _marca;

  Carro(super.color, super.velocidad, this._marca) {
    print(“Constructor hijo -  Carro”);
  }

  @override
  void showVehicle() {
    print(“Color: $color”);
    print(“Velocidad: $velocidad”);
    print(“Marca: $_marca”);
  }
}
```

Archivo: lib/Motocicleta.dart

```
import ‘Vehiculo.dart’;

class Motocicleta extends Vehiculo {
  String _modelo;

  Motocicleta(super.color, super.velocidad, this._modelo) {
    print(“Constructor hijo - Motocicleta”);
  }

  @override
  void showVehicle() {
    print(“Color: $color”);
    print(“Velocidad: $velocidad”);
    print(“Modelo: $_modelo”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Carro.dart’;
import ‘package:mi_app/Motocicleta.dart’;

void main(List<String> args) {
  var carro1 = Carro(“rojo”, 100, “Ford”);
  carro1.showVehicle();
  print(carro1.maxVelocidad());
  print(“-” * 20);

  var moto1 = Motocicleta(“Azul”, 60, “T1000”);
  moto1.showVehicle();
  print(moto1.maxVelocidad());
}
```

Salida:

```
Constructor padre - Vehiculo
Constructor hijo -  Carro
Color: rojo
Velocidad: 100
Marca: Ford
200
--------------------
Constructor padre - Vehiculo
```

```
Constructor hijo - Motocicleta
Color: Azul
Velocidad: 60
Marca: T1000
120
```

Ejemplo: Constructor nombrado en la clase Carro

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String? _color;
  int? _velocidad;

  Vehiculo(this._color, this._velocidad) {
    print(“Constructor padre - Vehiculo”);
  }
  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;

  void showVehicle() {
    print(“Color: $_color”);
    print(“Velocidad: $_velocidad”);
  }
}
```

Archivo: lib/Carro.dart

```
import ‘Vehiculo.dart’;

class Carro extends Vehiculo {
  String _marca;
  String? _tipo;

  Carro(super.color, super.velocidad, this._marca) {
    print(“Constructor hijo -  Carro”);
  }

  Carro.Tipo(super.color, super.velocidad, this._marca, this._
```

```
tipo) {
    print(“Constructor hijo -  Carro tipo: $_tipo”);
  }

  @override
  void showVehicle() {
    print(“Color: $color”);
    print(“Velocidad: $velocidad”);
    print(“Marca: $_marca”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Carro.dart’;

void main(List<String> args) {
  var carro1 = Carro.Tipo(“rojo”, 100, “Ford”, “Automático”);
  carro1.showVehicle();
  print(carro1.maxVelocidad());
  print(“-” * 20);
}
```

Salida:

```
Constructor padre - Vehiculo
Constructor hijo -  Carro tipo: Automático
Color: rojo
Velocidad: 100
Marca: Ford
200
```

Ejemplo: Invocación de métodos de la clase padre en la clase hija

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String? _color;
  int? _velocidad;

  Vehiculo(this._color, this._velocidad) {
    print(“Constructor padre - Vehiculo”);
  }
  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;
```

```
  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;

  void showVehicle() {
    print(“Color: $_color”);
    print(“Velocidad: $_velocidad”);
  }
}
```

Archivo: bin/Carro.dart

```
import ‘Vehiculo.dart’;

class Carro extends Vehiculo {
  String _marca;
  String? _tipo;

  Carro(super.color, super.velocidad, this._marca) {
    print(“Constructor hijo -  Carro”);
  }

  Carro.Tipo(super.color, super.velocidad, this._marca, this._
tipo) {
    print(“Constructor hijo -  Carro tipo: $_tipo”);
  }

  @override
  void showVehicle() {
    print(“Color: $color”);
    print(“Velocidad: $velocidad”);
    print(“Marca: $_marca”);
  }

  void showVehiclePadre() {
    super.showVehicle();
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Carro.dart’;

void main(List<String> args) {
  var carro1 = Carro.Tipo(“rojo”, 100, “Ford”, “Automático”);
  carro1.showVehicle();
  print(carro1.maxVelocidad());
  print(“-” * 20);

  carro1.showVehiclePadre();
}
```

Salida:

```
Constructor padre - Vehiculo
Constructor hijo -  Carro tipo: Automático
Color: rojo
Velocidad: 100
Marca: Ford
200
--------------------
Color: rojo
Velocidad: 100
```

Capítulo 12
Clases abstractas

Este capítulo se enfoca en las clases abstractas en Dart, exploraremos un concepto fundamental en la programación orientada a objetos. Las clases abstractas proporcionan un mecanismo para definir estructuras y comportamientos comunes que deben ser implementados por las clases hijas. A través de ejemplos concretos, examinaremos cómo diseñar y utilizar clases abstractas en Dart, y cómo crear instancias de la clase padre y sus clases hijas. Comprender cómo trabajar con clases abstractas es esencial para la organización y estructuración eficiente de nuestro código en Dart. Así que exploremos este aspecto crucial de la programación orientada a objetos, a continuación, te mencionamos algunas de sus características:

- No se pueden crear objetos con las clases abstractas.
- No se pueden instanciar.
- Se pueden heredar.

Ejemplo: Instancia de la clase padre vehículo, carro y motocicleta

Archivo: lib/Vehiculo.dart

```
class Vehiculo {
  String? _color;
  int? _velocidad;

  Vehiculo(this._color, this._velocidad);

  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;

  void showVehicle() {
    print(“Color: $_color”);
    print(“Velocidad: $_velocidad”);
  }
}
```

Archivo: lib/Carro.dart

```
import ‘Vehiculo.dart’;

class Carro extends Vehiculo {
  String _marca;
  String? _tipo;

  Carro(super.color, super.velocidad, this._marca);

  @override
  void showVehicle() {
    print(“Color: $color”);
    print(“Velocidad: $velocidad”);
    print(“Marca: $_marca”);
  }

  void showVehiclePadre() {
    super.showVehicle();
  }
}
```

Archivo: lib/Motocicleta.dart

```
import ‘Vehiculo.dart’;

class Motocicleta extends Vehiculo {
  String _modelo;

  Motocicleta(super.color, super.velocidad, this._modelo);

  @override
  void showVehicle() {
    print(“Color: $color”);
    print(“Velocidad: $velocidad”);
    print(“Modelo: $_modelo”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Carro.dart’;
import ‘package:mi_app/Motocicleta.dart’;
import ‘package:mi_app/Vehiculo.dart’;
```

```
void main(List<String> args) {
  Vehiculo vehiculo = Vehiculo(“rojo”, 50);
  Carro carro = Carro(“Azul”, 100, “Jeep”);
  Motocicleta moto = Motocicleta(“Amarillo”, 60, “T1000”);

  print(“---Vehiculo---”);
  vehiculo.showVehicle();
  print(“---Carro---”);
  carro.showVehicle();
  print(“---Motocicleta---”);
  moto.showVehicle();
}
```

Salida:

```
---Vehiculo---
Color: rojo
Velocidad: 50
---Carro---
Color: Azul
Velocidad: 100
Marca: Jeep
---Motocicleta---
Color: Amarillo
Velocidad: 60
Modelo: T1000
```

Ejemplo: Clase abstracta Vehiculo

Archivo: lib/Vehiculo.dart

```
abstract class Vehiculo {
  String? _color;
  int? _velocidad;

  Vehiculo(this._color, this._velocidad);

  set color(String color) => _color = color;
  set velocidad(int velocidad) => _velocidad = velocidad;

  String get color => _color!;
  int get velocidad => _velocidad!;

  int maxVelocidad() => _velocidad! * 2;
```

```
  void showVehicle() {
    print(“Color: $_color”);
    print(“Velocidad: $_velocidad”);
  }
}
```

Archivo: lib/Carro.dart

```
import ‘Vehiculo.dart’;

class Carro extends Vehiculo {
  String _marca;
  String? _tipo;

  Carro(super.color, super.velocidad, this._marca);

  @override
  void showVehicle() {
    print(“Color: $color”);
    print(“Velocidad: $velocidad”);
    print(“Marca: $_marca”);
  }

  void showVehiclePadre() {
    super.showVehicle();
  }
}
```

Archivo: lib/Motocicleta.dart

```
import ‘Vehiculo.dart’;

class Motocicleta extends Vehiculo {
  String _modelo;

  Motocicleta(super.color, super.velocidad, this._modelo);

  @override
  void showVehicle() {
    print(“Color: $color”);
    print(“Velocidad: $velocidad”);
    print(“Modelo: $_modelo”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Carro.dart’;
import ‘package:mi_app/Motocicleta.dart’;
//Unused import: ‘package:mi_app/Vehiculo.dart’.
//Try removing the import directive.
import ‘package:mi_app/Vehiculo.dart’;

void main(List<String> args) {
  //Vehiculo vehiculo = Vehiculo(“rojo”, 50);
  Carro carro = Carro(“Azul”, 100, “Jeep”);
  Motocicleta moto = Motocicleta(“Amarillo”, 60, “T1000”);

  print(“---Vehiculo---”);
  //vehiculo.showVehicle();
  print(“---Carro---”);
  carro.showVehicle();
  print(“---Motocicleta---”);
  moto.showVehicle();
}
```

Salida:

```
---Vehiculo---
---Carro---
Color: Azul
Velocidad: 100
Marca: Jeep
---Motocicleta---
Color: Amarillo
Velocidad: 60
Modelo: T1000
```

Capítulo 13
Mixins, interfaces y genéricos

Mixins

En el contexto de la programación en Dart, los mixins representan una herramienta poderosa para compartir funcionalidades entre clases de una manera flexible y reutilizable. Los mixins permiten que múltiples clases compartan métodos y propiedades, sin necesidad de herencia de una clase base común. En esta sección, exploraremos a fondo el uso de mixins en Dart y cómo pueden mejorar el modularidad y la reutilización de código en nuestras aplicaciones. A través de ejemplos prácticos, descubriremos cómo implementar y aprovechar esta característica para optimizar nuestro flujo de desarrollo y construir aplicaciones más eficientes y mantenibles.

Referencia

Ejemplo:

```
void main(List<String> args) {
  Dolphin dolphin = Dolphin();
  dolphin.swim();

  Duck duck = Duck();
  duck.fly();
  duck.swim();
  duck.walk();

  Shark shark = Shark();
  shark.swim();
}

class Animal {}

class Mammal extends Animal {}

class Bird extends Animal {}
```

```
class Fish extends Animal {}

class Dolphin extends Mammal with Swimmer {
  Dolphin() {
    print(“---Dolphin---”);
  }
}

class Duck extends Bird with Walker, Flyer, Swimmer {
  Duck() {
    print(“---Duck---”);
  }
}

class Shark extends Fish with Swimmer {
  Shark() {
    print(“---Shark---”);
  }
}

abstract class Walker {
  void walk() => print(‘Walking...’);
}

abstract class Flyer {
  void fly() => print(‘Flying...’);
}

abstract class Swimmer {
  void swim() => print(‘swimming...’);
}
```

Salida:

```
---Dolphin---
swimming...
---Duck---
Flying...
swimming...
Walking...
---Shark---
swimming...
```

Interfaces

En la programación en Dart, las interfaces desempeñan un papel crucial en la definición de contratos y la especificación de comportamientos que las clases deben seguir. Aunque Dart no tiene una sintaxis específica para interfaces como en otros lenguajes, se puede lograr el mismo efecto utilizando clases abstractas. En esta sección, exploraremos cómo utilizar clases abstractas para implementar interfaces en Dart, lo que nos permite definir comportamientos comunes para varias clases. A través de ejemplos detallados, exploraremos cómo aplicar e implementar interfaces, brindando flexibilidad y coherencia en nuestra programación en Dart, a continuación, algunas características:

- Permite la reutilización de código.
- Simular la herencia múltiple.

Ejemplo: Implementación de la clase ControlRemoto y Televisor

Archivo: lib/ControlRemoto.dart

```
class ControlRemoto {
  void subirVolumen() {
    print(“subiendo volumen”);
  }

  void bajarVolumen() {
    print(“bajando volumen”);
  }
}
```

Archivo: lib/Televisor.dart

```
import ‘package:mi_app/ControlRemoto.dart’;

class Televisor implements ControlRemoto {
  @override
  void bajarVolumen() {
    print(“subiendo volumen en el Televisor”);
  }

  @override
  void subirVolumen() {
    print(“bajando volumen en el Televisor”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/ControlRemoto.dart’;
import ‘package:mi_app/Televisor.dart’;

void main(List<String> args) {
  Televisor t = Televisor();

  t.subirVolumen();
  t.bajarVolumen();

  ControlRemoto cr = ControlRemoto();
  cr.subirVolumen();
  cr.bajarVolumen();
}
```

Salida:

```
bajando volumen en el Televisor
subiendo volumen en el Televisor
subiendo volumen
bajando volumen
```

Ejemplo: Implementación de la clase abstracta ControlRemoto y las clases Televisor y Radio

Archivo: lib/ControlRemoto.dart

```
abstract class ControlRemoto {
  void subirVolumen() {}

  void bajarVolumen() {}
}
```

Archivo: lib/Televisor.dart

```
import ‘ControlRemoto.dart’;

class Televisor implements ControlRemoto {
  @override
  void bajarVolumen() {
    print(“subiendo volumen en el Televisor”);
  }

  @override
  void subirVolumen() {
```

```
    print(“bajando volumen en el Televisor”);
  }
}
```

Archivo: lib/Radio.dart

```
import ‘ControlRemoto.dart’;

class Radio implements ControlRemoto {
  @override
  void bajarVolumen() {
    print(“subiendo volumen al Radio”);
  }

  @override
  void subirVolumen() {
    print(“bajando volumen al Radio”);
  }
}
```

Archivo: bin/mi_app.dart

```
import ‘package:mi_app/Televisor.dart’;
import ‘package:mi_app/Radio.dart’;

void main(List<String> args) {
  Televisor t = Televisor();
  t.subirVolumen();
  t.bajarVolumen();

  Radio r = Radio();
  r.subirVolumen();
  r.bajarVolumen();
}
```

Salida:

```
bajando volumen en el Televisor
subiendo volumen en el Televisor
bajando volumen al Radio
subiendo volumen al Radio
```

Genéricos (generics)

Los genéricos permiten crear código que puede trabajar con distintos tipos de datos, brindando flexibilidad y reutilización de código en una variedad de situaciones. Exploraremos cómo definir clases y métodos genéricos, así como su aplicación en listas y funciones, proporcionando una comprensión sólida de cómo aprovechar al máximo este potente recurso en Dart. A lo largo de ejemplos prácticos y ejercicios, desentrañaremos los beneficios y la implementación adecuada de genéricos en Dart, enriqueciendo nuestro conjunto de herramientas para construir programas robustos y adaptables.

Ejemplo: Clase con propiedades dinámicas (dynamic)

```
void main(List<String> args) {
  Persona hugo = Persona(111, “Hugo”);
  hugo.showData();

  Persona paco = Persona(“111”, “Paco”);
  paco.showData();
}

class Persona {
  dynamic _id;
  String _nombre;
  Persona(this._id, this._nombre);

  void showData() {
    print(“ID: $_id”);
    print(“Nombre: $_nombre”);
  }
}
```

Salida:

```
ID: 111
Nombre: Hugo
ID: 111
Nombre: Paco
```

Ejemplo: Clase con genéricos (generics)

```
void main(List<String> args) {
  Persona<int> hugo = Persona<int>(111, “Hugo”);
  hugo.showData();

  Persona<String> paco = Persona<String>(“111”, “Paco”);
```

```
  paco.showData();
}

class Persona<T> {
  T _id;
  String _nombre;
  Persona(this._id, this._nombre);

  void showData() {
    print("ID: $_id");
    print("Nombre: $_nombre");
  }
}
```

Salida:

```
ID: 111
Nombre: Hugo
ID: 111
Nombre: Paco
```

Ejemplo: Listas con genéricos

```
void main(List<String> args) {
  List<int> numeros = [];
  numeros.addAll([1, 2, 4, 8]);
  print(numeros);

  List<String> nombres = [];
  nombres.addAll(["Hugo", "Paco", "Luis"]);
  print(nombres);
}
```

Salida:

```
[1, 2, 4, 8]
[Hugo, Paco, Luis]
```

Ejemplo: Métodos genéricos

- Lugares donde se puede aplicar:
  1. En los argumentos
  2. En las variables locales del método
  3. En el retorno

Versión 1

```
void main(List<String> args) {
  print(primerElemento([1, 2, 3, 4, 5, 6]));
  print(primerElemento([“Hugo”, “Paco”, “Luis”]));
  print(primerElemento([3.14, “Paco”, 5]));
  print(primerElemento([true, “Paco”, “Luis”]));
  print(primerElemento([
    [1, 2, 3],
    “Paco”,
    “Luis”
  ]));
}

T primerElemento<T>(List<T> lista) {
  T elemento = lista[0];
  return elemento;
}
```

Salida:

```
1
Hugo
3.14
true
[1, 2, 3]
```

Versión 2

```
void main(List<String> args) {
  List listaInt = <int>[1, 2, 3, 4, 5, 6];
  print(primerElemento(listaInt));

  List listaString = <String>[“Hugo”, “Paco”, “Luis”];
  print(primerElemento(listaString));

  List listaDynamic = <dynamic>[3.14, “Paco”, 5];
  print(primerElemento(listaDynamic));

  listaDynamic = [true, “Paco”, “Luis”];
  print(primerElemento(listaDynamic));
  listaDynamic = [
    [1, 2, 3],
    “Paco”,
    “Luis”
```

```
  ];
  print(primerElemento(listaDynamic));
}

T primerElemento<T>(List<T> lista) {
  T elemento = lista[0];
  return elemento;
}
```

Salida:

```
1
Hugo
3.14
true
[1, 2, 3]
```

Ejemplo: Genéricos restringidos (extend), cuadrado de un número

```
void main(List<String> args) {
  cuadrado(3);
  cuadrado(3.5);
  //cuadrado("Hola");

  cuad(3);
  cuad(3.5);
  //cuad("Hola");
}

void cuadrado(dynamic n) {
  print(n * n);
}

cuad<T extends num>(T n) {
  print(n * n);
}
```

Salida:

```
9
12.25
9
12.25
```

Ejemplo: Genéricos restringidos (extend), producto de dos números

```
void main(List<String> args) {
  prod(3, 4);
  prod(3.5, 6.7);
  prod(2, 5.2);
  //prod(“Hola”, “ Mundo”);

  prodGeneric(3, 4);
  prodGeneric(3.5, 6.7);
  prodGeneric(2, 5.2);
  //prodGeneric(“Hola”, “ Mundo”);
}

prod(dynamic n1, dynamic n2) {
  print(n1 * n2);
}

prodGeneric<T extends num>(T n1, T n2) {
  print(n1 * n2);
}
```

Salida:

```
12
23.45
10.4
12
23.45
10.4
```

Ejemplo: Métodos/funciones genéricas con retorno de datos, suma de dos números

```
void main(List<String> args) {
  print(sumaGeneric(3, 4));
  print(sumaGeneric(3.5, 6.7));
  print(sumaGeneric(2, 5.2));
  //sumaGeneric(“Hola”, “ Mundo”);
}

num sumaGeneric<T extends num>(T n1, T n2) {
  return n1 + n2;
}
```

Salida:

```
7
10.2
7.2
```

Ejemplo: Sumar un número con la suma de una lista de números y multiplicación de dos números

```
void main(List<String> args) {
  print(sumar<int>(3, [1, 2, 3, 4, 5]));
  print(sumar(3, [1, 3.5, 3, 4.10, 5]));

  print(multiply<int>(5, 6));
  print(multiply(5, 6));
}

T sumar<T extends num>(T n, List<T> numeros) {
  var suma = n;

  for (var numero in numeros) {
    suma = (suma + numero) as T;
  }
  return suma;
}

T multiply<T extends num>(T a, T b) {
  return (a * b) as T;
}
```

Salida:

```
18
19.6
30
30
```

Ejemplo: Clases genéricas

```
void main(List<String> args) {
  Contador<double> doubles = Contador<double>();
  doubles.addAll([1.1, 1.2, 1.3]);
  doubles.total();

  Contador<int> ints = Contador<int>();
```

```
  ints.addAll([1, 2, 3]);
  ints.total();
}

class Contador<T extends num> {
  List _items = <T>[];

  void addAll(Iterable<T> iterable) => _items.addAll(iterable);
  void add(T value) => _items.add(value);

  T elementAt(int index) => _items.elementAt(index);

  void total() {
    num value = 0;

    for (var val in _items) {
      value += val as T;
    }
    print(value);
  }
}
```

Salida:

```
3.5999999999999996
6
```

Capítulo 14
Programación asíncrona

En este apartado, exploraremos un aspecto crucial de la programación en Dart: la programación asíncrona. En muchos escenarios, especialmente al trabajar con operaciones de entrada y salida o tareas que pueden tardar un tiempo significativo, es esencial mantener la eficiencia y la capacidad de respuesta de nuestros programas. A través de ejemplos y casos de uso, aprenderemos cómo utilizar constructores asíncronos, como Futures, para gestionar tareas de manera asíncrona y aprovechar al máximo el rendimiento de nuestras aplicaciones. Además, veremos cómo enfrentar situaciones donde la ejecución síncrona no es suficiente, y cómo implementar lógica asíncrona para mejorar la experiencia del usuario y la eficiencia de nuestros programas en Dart.

- Sirve para ejecutar aplicaciones en segundo plano.

Ejemplo: Programa síncrono

```
import ‘dart:io’;

void main(List<String> args) {
  stdout.write(“Dame tu nombre: “);
  String? nombre = stdin.readLineSync();
  print(“Hola $nombre”);
}
```

Salida:

```
Dame tu nombre: Walter
Hola Walter
```

Ejemplo: Programa asíncrono que lee un archivo de texto y lo despliega

Archivo bin/contactos.txt

```
1. Hugo
2. Paco
3. Luis
4. Maria
5. Karla
```

Archivo: bin/mi_app.dart

```
import ‘dart:io’;
import ‘dart:async’;
```

```
void main(List<String> args) {
  File file = File(Directory.current.path + “/bin/contactos.txt”);

  Future<String> f = file.readAsString();

  f.then((value) => print(value));

  print(“Fin del programa”);
}
```

Salida:

```
Fin del programa
1. Hugo
2. Paco
3. Luis
4. Maria
5. Karla
```

Ejemplo: Future

- Es el resultado de una operación asíncrona.
- Puede tener dos estados:
  - Incompleto
  - Completo

```
import ‘dart:async’;

void main(List<String> args) {
  print(“Iniciando programa...”);
  saludo(“Probando un Future”).then((value) => print(value));
  print(“Finalizando programa...”);
}

Future<String> saludo(String msg) {
  return Future.delayed(Duration(seconds: 5), () => “Hola,
$msg”);
}
```

Salida:

```
Iniciando programa...
Finalizando programa...
Hola, Probando un Future
```

Conclusiones de la parte II

En esta extensa sección dedicada a la Programación Orientada a Objetos (POO) con Dart, hemos recorrido un viaje informativo y práctico a través de diversos aspectos fundamentales de la programación orientada a objetos. Desde la conceptualización básica de las clases hasta la aplicación avanzada de conceptos como herencia, clases abstractas, mixins, interfaces y genéricos, hemos explorado cómo aplicar estos principios en el contexto del lenguaje Dart.

Comenzando con la esencia de las clases y sus componentes, hemos aprendido a definir propiedades y métodos, así como a crear instancias de clases y a utilizarlas eficazmente en nuestras aplicaciones. Los constructores, tanto básicos como nombrados, fueron introducidos para permitir una inicialización efectiva de objetos.

La herencia, un concepto clave en POO, se abordó detalladamente, mostrando cómo una clase puede heredar propiedades y comportamientos de otra, facilitando la reutilización del código y la creación de jerarquías de clases. Asimismo, exploramos cómo las clases abstractas proporcionan una estructura común para las subclases y cómo los mixins ofrecen una forma de compartir comportamientos entre distintas clases.

No obstante, no nos detuvimos en esto, sino que avanzamos hacia la implementación de interfaces y genéricos, brindando flexibilidad y reutilización aún mayor en nuestros programas. Cerrando esta parte abordamos la programación asíncrona, fundamental para aplicaciones modernas y eficientes que necesitan manejar múltiples tareas concurrentes.

Finalmente, consideramos que hemos proporcionado un material de utilidad para la comprensión de la Programación Orientada a Objetos en Dart y cómo se aplica en diversos contextos. Estos conocimientos son esenciales para cualquier desarrollador que busque crear aplicaciones robustas y mantenibles en Dart, que seguramente establecerán una base sólida para el desarrollo futuro en áreas avanzadas de este lenguaje de programación.

Referencias

Buckett, C. (2013). *Dart in action*. Manning Publications. https://www.manning.com/books/dart-in-action

Walrath, K. y Ladd, S. (2012). *Dart: Up and running*. O'Reilly Media, Inc. https://books.google.com.mx/books/about/Dart.html?id=xLxqmzlZHU4C&redir_esc=y

Kopec, D. (2014). *Dart for absolute beginners*. Apress. https://doi.org/10.1007/978-1-4302-6482-8

Akopkokhyants, S. (2014). *Mastering Dart: Community experience distilled*. Packt Publishing. https://books.google.com.mx/books/about/Mastering_Dart.html?id=4p60rQEACAAJ&redir_esc=y

Balbaert, I. (2014). *DART cookbook: Over 110 incredibly effective, useful, and hands-on recipes to design Dart web client and server applications*. Packt Publishing. https://www.packtpub.com/en-us/product/dart-cookbook-9781783989621

Reseñas curriculares

Walter Alexander Mata López
ORCID: 0000-0002-8107-2182
Ingeniero en sistemas computacionales, maestro en ciencias área computación, diploma en estudios avanzados de tercer ciclo en métodos y técnicas avanzadas de desarrollo de software por la Universidad de Granada, España, doctor en socioformación y sociedad del conocimiento por el Centro Universitario CIFE, México. Doctor en tecnología educativa por el Centro Escolar Mar de Cortés, México. Tiene el reconocimiento SNII-I por la SECIHTI de México. Es autor de libros, capítulos de libro y artículos publicados en congresos y revistas internacionales. Su área de interés es la aplicación de la tecnología educativa en la ingeniería, inteligencia artificial y ciencia de datos.

Mónica Cobián Alvarado
ORCID: 0009-0007-8441-0873
Ingeniera en sistemas computacionales, maestra en ciencias área computación, diploma en estudios avanzados de tercer ciclo en métodos y técnicas avanzadas de desarrollo de software por la Universidad de Granada, España. Doctora en educación con más de 20 años de experiencia docente impartiendo materias como: matemáticas discretas, análisis y diseño de sistemas, seminario de investigación, entre otras. Su área de interés es la aplicación de la tecnología educativa en la ingeniería, ingeniería de software, entre otras áreas.

Armando Román Gallardo
ORCID: 0000-0001-5912-4428
Profesor-investigador de tiempo completo en la Facultad de Telemática de la Universidad de Colima. Ingeniero en sistemas computacionales con maestría en ciencias computacionales por la Universidad de Colima y doctorado en educación por la Universidad de Baja California. Sus intereses de investigación se centran en los procesos de desarrollo de software.

José Román Herrera Morales
ORCID: 0000-0003-0811-4187
Profesor-Investigador de tiempo completo en la Facultad de Telemática de la Universidad de Colima. Cursó la carrera de ingeniería en comunicaciones y electrónica, y la maestría en ciencias, área telemática, por la Universidad de Colima. Realizó su doctorado en tecnologías de la información en la Universidad de Guadalajara. Sus intereses de investigación incluyen los sistemas de búsqueda y recuperación de información, los sistemas inteligentes, la tecnología web, el procesamiento de datos, minería y ciencia de datos, entre otros.

Programación estructurada y orientada a objetos con Dart, de Walter Alexander Mata López, Mónica Cobián Alvarado, Armando Román Gallardo y José Román Herrera Morales, fue editado en la Dirección General de Publicaciones de la Universidad de Colima, avenida Universidad 333, Colima, Colima, México, www.ucol.mx. La edición se terminó en febrero de 2026. En la composición tipográfica se utilizó la familia Calibri y Consolas. Programa Editorial No Periódico: Eréndira Cortés Ventura. Diseño de portada: Dulce María Silva Novela. Corrección: Gabriel Govea Acosta. Diseño de interiores: José Luis Ramírez Moreno. Cuidado de la edición: Eréndira Cortés Ventura. Plataformas digitales: Benjamín Cortés Vega y Damara Josselin Jiménez Armenta.

La programación es una habilidad esencial de la era digital en la que vivimos. Cada vez más, las industrias y empresas de todo tipo requieren personal capacitado que programe y desarrolle software. El libro *Programación estructurada y orientada a objetos con Dart*, tiene como objetivo proporcionar una base sólida en programación utilizando Dart, un lenguaje de programación moderno y versátil que se ha convertido en una elección popular para el desarrollo de aplicaciones web y móviles. A lo largo de estas páginas, las y los estudiantes encontrarán un enfoque claro y estructurado para aprender los fundamentos de la programación con Dart, desde conceptos básicos hasta temas más avanzados.

ISBN 978-968-9733-20-1

UNIVERSIDAD DE COLIMA