Dominando Dart: Desde la Amazonia

EDITORIAL CIIDES

Dominando Dart:

Desde la Amazonia

Autores:

Oscar Fabian Patiño Perdomo

Doctor en Educación, Magister en Gerencia de Sistemas de Información y Proyectos Tecnológicos y Especialista en Gestión de Sistemas y Tecnologías de la Información en la Empresa; Docente de la Universidad de la Amazonia, Facultad de ingeniería, Centro de investigación Innovación y desarrollo para la sustentabilidad S.A.S BIC/Group of Research in Engineering and Nature of Amazonia University (GREEN)/Semillero de Investigación R3iinova, ORCID: 0000-0001-6433-0402, Florencia, Colombia, o.patino@udla.edu.co

Wilmer Arley Patiño Perdomo

Doctor en Gerencia y Política Educativa, Especialista en Ingeniería de Software, Docente de la Universidad de la Amazonia, Facultad de ingeniería, Centro de investigación Innovación y desarrollo para la sustentabilidad S.A.S BIC /Semillero Enjambre, ORCID: 0000-0002-4338-6543, Florencia, Colombia, w.patino@udla.edu.co

Franklin Gamboa Tabares

Magister en Educación, Docente de la Universidad de la Amazonía, Facultad de Educación, Centro de investigación Innovación y desarrollo para la sustentabilidad S.A.S BIC, ORCID: 0000-0003-3059-2062, Florencia-Caquetá, Colombia, f.gamboa@udla.edu.co.

Libro de texto producto de la articulación de docentes del Programa de Ingeniería de Sistemas, Departamento de Pedagogía de la Universidad de la Amazonia y el Centro de investigación Innovación y desarrollo para la sustentabilidad S.A.S BIC

Dominando Dart: Desde la Amazonia

Incluye bibliografía.

66 páginas

ISBN (Digital): 978-628-95694-8-3

Número y año de edición: Primera edición, septiembre 2025.
Diseño de cubierta: Copilot (IA) mediante Prompt Oscar Fabián Patiño.
Diagramación y terminación: Editorial CIIDES.
Como citar: Patiño et al. (2025). *Dominando Dart: Desde la Amazonia.* Editorial CIIDES.

Centro de investigación, innovación y Desarrollo para la Sustentabilidad – CIIDES SAS BIC.
NIT: 901.629.503-2
Editorial CIIDES

Primera edición, Florencia (Colombia), 2025.
Coordinadora Editorial: Leidy Katherine Losada Claros

EDITORIAL CIIDES

Este libro de texto es el resultado de la sinergia entre docentes del Programa de Ingeniería de Sistemas, Departamento de Pedagogía de la Universidad de la Amazonia y el Centro de investigación Innovación y desarrollo para la sustentabilidad S.A.S BIC, con una finalidad pedagógica para la enseñanza de los conceptos básicos del lenguaje de programación Dart. Este texto fue evaluado en la modalidad de doble ciego y contó con una evaluación editorial.

COMITÉ ACADÉMICO, EDITORIAL Y DE INVESTIGACIÓN

TABLA DE CONTENIDO

1 RESUMEN

Este libro ofrece una guía completa y detallada sobre el lenguaje de programación Dart, desarrollado por Google. Se exploran sus aplicaciones en el desarrollo de aplicaciones web, móviles y de escritorio, destacando su compatibilidad con el framework Flutter. A través de ejemplos prácticos y explicaciones sencillas, se introduce a los lectores en los conceptos fundamentales de Dart, su sintaxis, y las mejores prácticas para su uso. Además, se abordan técnicas avanzadas y características del lenguaje, como la compilación ahead-of-time (AOT) y just-in-time (JIT), mixins, y métodos de extensión, proporcionando una visión integral para desarrolladores de todos los niveles.

Palabras clave: Dart, Flutter, Programación Orientada a Objetos, Desarrollo Web, Compilación

2 ABSTRACT

This book provides a comprehensive and detailed guide to the Dart programming language, developed by Google. It explores its applications in web, mobile, and desktop application development, highlighting its compatibility with the Flutter framework. Through practical examples and simple explanations, readers are introduced to the fundamental concepts of Dart, its syntax, and best practices for its use. Additionally, advanced techniques and language features such as ahead-of-time (AOT) and just-in-time (JIT) compilation, mixins, and extension methods are covered, offering a holistic view for developers of all levels.

Keywords: Dart, Flutter, Object-Oriented Programming, Web Development, Compilation

3 PRÓLOGO

El lenguaje de programación Dart, desarrollado por Google, se ha constituido como un instrumento potente y eficaz para la creación de aplicaciones web, móviles y de escritorio. Su vinculación con el marco de trabajo Flutter ha potenciado su popularidad, convirtiéndolo en una de las alternativas preferidas por los programadores.

Este libro es un manual práctico para aquellos que desean adquirir conocimientos de Dart, desde los fundamentos hasta métodos más sofisticados. Como autores, nos hemos enfocado en el progreso de Dart y Flutter, con el objetivo de compartir nuestro conocimiento con la comunidad de programadores. En contenido, encontraran explicaciones simples, ejemplos valiosos y recomendaciones para maximizar el uso de este lenguaje.

Nuestra meta no solo consiste en enseñarles a programar en Dart, sino también en incentivarle a explorar nuevas alternativas e innovar en la creación de software.

4 INTRODUCCIÓN A DART

Dart es un lenguaje de programación que fue creado por Google y se usa principalmente para hacer aplicaciones en línea, móviles y de escritorio. Dart es un lenguaje de programación que se enfoca en objetos. Su sintaxis es similar a la de Java y C#, lo que lo hace fácil de aprender para los desarrolladores debido a que muy posiblemente ya conocen estos lenguajes populares.

Dart se usa principalmente para crear aplicaciones en la web y en dispositivos móviles, gracias a su compatibilidad con Flutter; el marco de trabajo de Google para el desarrollo de aplicaciones móviles. Flutter ayuda a los programadores a crear aplicaciones personalizadas para iOS y Android usando el lenguaje Dart.

Además, Dart se usa de igual forma para crear aplicaciones web, debido a que tiene un compilador que convierte el código Dart en código JavaScript. Esto permite que a su vez las aplicaciones escritas en Dart funcionen en navegadores web.

Así mismo, Dart cuenta con muchas bibliotecas y herramientas de desarrollo. Por ejemplo, AngularDart es un marco para crear aplicaciones web, y DartPad es una herramienta en línea para escribir y ejecutar código (Dart *Tools*, s. f.).

En resumen, Dart es un lenguaje de programación flexible y fácil de aprender. Se usa sobre todo para hacer aplicaciones web, móviles y de escritorio, y cuenta con muchas herramientas y frameworks que facilitan su desarrollo aparte de contar con una curva rápida de aprendizaje por aquellos que tienen nociones de programación en lenguajes como Java, C#, JavaScript entre otros.

4.1 Breve historia de dart

Dart fue creado en 2011 por los ingenieros Lars Bak y Kasper Lund para facilitar un lenguaje de programación moderno, eficiente y fuerte para el desarrollo de aplicaciones web y móviles. Desde el principio, se creó como un lenguaje orientado a objetos, con una estructura que permite una alta escalabilidad y un alto rendimiento, características que lo hace relevante frente a otros lenguajes actuales. La claridad y organización de su estructura semántica, junto con su enfoque en la facilidad de lectura y mantenimiento del código, lo establecieron rápidamente como una opción prioritaria en el desarrollo de software especialmente en lo que al enfoque móvil se refiere (Google for Developers, 2012).

Una de las etapas más importantes en el desarrollo de Dart fue su adopción como el lenguaje principal de Flutter, el marco de Google para crear interfaces de usuario nativas para dispositivos basados en Android y iOS. Esta integración hizo que el desarrollo fuera más sencillo, debido a que Dart se puede usar tanto en el cliente como en el servidor, mejorando la eficiencia del proceso.

A lo largo de su desarrollo, Dart ha visto grandes avances gracias a la gran comunidad de desarrolladores con los que cuenta generando nuevas dependencias y al respaldo de Google como proyecto de código abierto y todo el talento humano y capital detrás de este. Su uso se ha ampliado a

muchas áreas, desde el desarrollo web hasta las aplicaciones para el Internet de las Cosas (IoT), demostrando su flexibilidad en el desarrollo de software.
Hoy en día, Dart es una opción muy competitiva para crear aplicaciones de alto rendimiento en tiempos relativamente cortos. Su enfoque en la programación orientada a objetos, su creciente ecosistema y su conexión y adaptación con tecnologías actuales hacen de este lenguaje uno muy importante para quienes quieren buscan una solución o desean manejar desarrollos multiplataformas.

4.2 Dart2js

Dart2js es un programa que transforma el código Dart en código JavaScript. El objetivo de Dart2js es hacer que el código escrito en Dart funcione en navegadores web que no tienen un entorno de ejecución de Dart.

En consecuencia, Dart2js al lograr transformar el código Dart en JavaScript de forma eficiente, también permite que el código generado funcione casi igual que en un entorno nativo de Dart. Esto quiere decir que el código creado por Dart2js puede usar las funciones del navegador y además puede trabajarse junto con otras bibliotecas y marcos de JavaScript.

Dart2js se usa junto con Flutter, que es un marco de desarrollo móvil de Google, para hacer aplicaciones web con Flutter. Con Dart2js, los desarrolladores de Flutter pueden escribir código una vez y usarlo en dispositivos móviles y en la web.

Como instancia final el código creado por Dart2js puede ser un poco más lento y también más grande debido a cómo es JavaScript. Sin embargo, hay herramientas y técnicas que pueden ayudar a mejorar esto.

4.3 Dart2native

Dart2Native es un compilador que transforma el código escrito en Dart a código nativo. Así pues, el objetivo de Dart2Native es permitir que el código escrito en Dart funcione de forma nativa en los diferentes dispositivos que soporta, sin necesidad de un entorno de ejecución de Dart (*dart2native*, s.f.).

Dart2Native puede transformar el código Dart en código nativo para varias plataformas, incluyendo Linux, Windows, MacOS, y también para dispositivos móviles basados en Android y iOS. Esto significa que el código creado por Dart2Native puede usar las funciones de la plataforma de forma nativa, lo que mejora el rendimiento y permite acceder a funciones específicas de cada plataforma.

En tal sentido Dart2Native se utiliza junto con Flutter, el marco de desarrollo de aplicaciones móviles de Google, para crear aplicaciones nativas con Flutter. Con Dart2Native, los desarrolladores de Flutter pueden escribir código una vez y usarlo en dispositivos móviles y en computadoras, aprovechando al máximo las funciones de las plataformas de una manera casi orgánica.

Es de resaltar que Dart2Native es una tecnología nueva y aún se está desarrollando, pero cada vez se vuelve más estable y mejora con rapidez debido al respaldo tecnológico con el que cuenta.

4.4 Just-in-time

Just-in-time (JIT) constituye un método de compilación en el que el código se compila durante la ejecución, en contraposición a su compilación previa a la misma. En lo que respecta a Dart, el motor que ejecuta el código cuenta con un compilador JIT (Just-In-Time), que transforma el código Dart en código de máquina nativo durante su ejecución (Dart overview. s. f. -a).

La interfaz de compilación JIT de Dart presenta múltiples beneficios en comparación con un compilador AOT (Ahead-of-Time).

- **Aumento de la flexibilidad**: El compilador JIT tiene la capacidad de adaptarse a las condiciones de ejecución en tiempo real, lo cual optimiza el desempeño en caso de modificaciones en el entorno de ejecución.

- **Minimización del tiempo de compilación**: El compilador Just In Time (JIT) solo requiere compilar el código que se está ejecutando en ese instante, en lugar de compilar el código completo antes de su ejecución.

- **Archivo de menor tamaño**: El código compilado por el método Just In Time (JIT) solo contiene el código requerido para su operación.

4.5 Ahead-of-time

Ahead-of-time (AOT) representa un procedimiento de compilación en el que el código se compila previamente a su ejecución. Con respecto al código Dart, el compilador AOT (Ahead-of-Time) transforma el código Dart en código nativo de máquina previo a su ejecución. En otras palabras, el código fuente se compila en un archivo ejecutable que alberga el código ya compilado y preparado para su ejecución en la plataforma de interés (Dart overview. s. f. -b).

El compilador AOT de Dart se usa para crear aplicaciones nativas para dispositivos móviles, como Android e iOS, y también para hacer aplicaciones de escritorio en Windows, MacOS y Linux. Esto permite a los desarrolladores hacer aplicaciones nativas usando Dart y Flutter, y aprovechar al máximo las funciones de la plataforma.

Algunas de las ventajas inherentes al uso de AOT incluyen:

Incremento en la eficiencia: El código compilado de manera AOT se ejecuta con mayor rapidez, dado que ya ha sido compilado y optimizado para la plataforma objetivo.

- **Mayor seguridad**: El código compilado en forma de AOT presenta una mayor complejidad y lo que incrementa la seguridad de la aplicación.

- **Archivo más pequeño**: El código compilado de forma AOT suele ser más pequeño debido a la optimización y eliminación de código no utilizado.

Es importante tener en cuenta que el procedimiento de compilación AOT es más complejo y demanda un conocimiento más avanzado de las plataformas de interés. Adicionalmente, el código compilado de manera AOT no puede ser modificado durante el tiempo de ejecución, al contrario del método JIT.

4.6 Camel case

Camel case es una convención de escritura para combinar varias palabras en una sola, en la cual cada palabra comienza con una letra mayúscula, excepto la primera, y se unen sin espacios. La convención se llama "camel case" debido a que la línea divisoria entre las palabras forma una serie de jorobas o lomos de camello (*Effective Dart: Style*, s.f.).

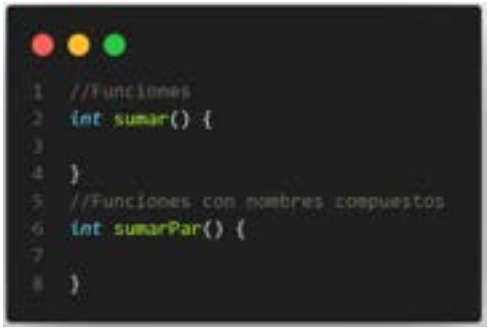
Existen dos tipos de Camel Case, "**UpperCamelCase**" o "**Pascal Case**": También conocido como **PascalCase**, es un estilo de escritura en el que cada palabra en el identificador comienza con una letra mayúscula y no hay espacios entre las palabras. Este estilo se usa comúnmente para nombrar **clases**.

Ahora bien, si el nombre de la clase está compuesto de más de una palabra, cada una de estas se debe comenzar con mayúscula.

"**lowerCamelCase**": También conocido como camelCase, es un estilo de escritura en el que la primera palabra comienza con una letra minúscula y cada palabra subsiguiente comienza con una letra mayúscula, sin espacios entre las palabras. Este estilo se usa comúnmente para nombrar **variables**, **métodos** y **funciones**.

Ahora bien, de manera similar aplica para los métodos.

Al igual que las funciones.



Camel case es ampliamente utilizado en programación, especialmente en lenguajes como Java, C#, Python, JavaScript y Dart. Es una convención de codificación que ayuda a mejorar la legibilidad del código y facilita la lectura de variables, funciones y nombres de clases.

4.6.1 case

El snake case es una convención de nomenclatura adoptada por Dart para nombrar archivos y se encuentra dentro de sus reglas linter (*File_Names*, s.f.). En este formato, los nombres se escriben en minúsculas y, cuando contienen múltiples palabras, se separan con un guion bajo.

En resumen, las principales características de esta nomenclatura son:

- Todos los caracteres están en minúsculas.
- Las palabras se separan con guiones bajos cuando es necesario.
- No se utilizan espacios ni mayúsculas.

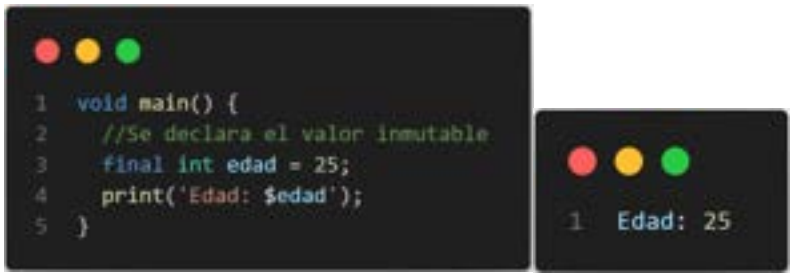
4.7 Estado de una Variable

Al momento de declarar variables se puede usar la palabra clave **final**, **const** o **late**, sin embargo, cada una tiene su propio comportamiento *Variables.* (s. f.).

4.7.1 Final

La palabra clave **final** indica que una variable es inmutable, por lo que su valor solo puede asignarse una vez, ya sea en el momento de la declaración o durante la ejecución.

En el siguiente ejemplo se evidencia al declararse.



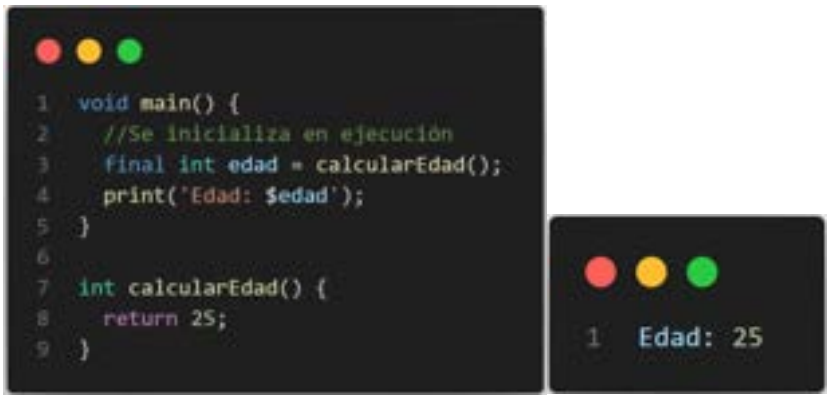
```dart
void main() {
  //Se declara el valor inmutable
  final int edad = 25;
  print('Edad: $edad');
}
```

```
Edad: 25
```

Nota: Si intenta después de inicializar la variable cambiar su valor generara un error.

Asimismo, el valor puede asignarse en tiempo de ejecución desde una función, una API, la resolución de la pantalla de un dispositivo, entre otros.

En el siguiente ejemplo se evidencia la asignación en tiempo de ejecución:



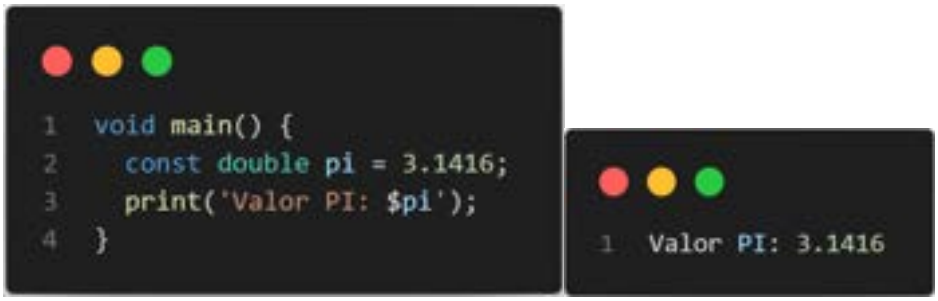
```dart
void main() {
  //Se inicializa en ejecución
  final int edad = calcularEdad();
  print('Edad: $edad');
}

int calcularEdad() {
  return 25;
}
```

```
Edad: 25
```

4.7.2 Const

La palabra clave **const** indica que el valor de la variable es conocido desde el momento de su creación y además es un inmutable.

Así como se puede observar en el siguiente ejemplo:



```dart
void main() {
  const double pi = 3.1416;
  print('Valor PI: $pi');
}
```

```
Valor PI: 3.1416
```

Nota: De manera similar a **final**, si se intenta cambiar el valor de la variable definida como **const** después de inicializarla, se generará un error.

4.7.3 Late

Una variable marcada como **late** no se inicializa en el momento de su creación, pero debe tener un valor asignado antes de su primer uso.

Así como se puede observar en el siguiente ejemplo:

```dart
void main() {
  late String titulo;
  // Se inicializa para luego ser usada
  titulo = 'Desde la Amazonia';
  print('Titulo: $titulo');
}
```

```
Titulo: Desde la Amazonia
```

Nota: En algunos casos, la palabra clave **late** puede ser reemplazada por el operador **?** de Sound Null Safety. Más adelante se abordará el uso de este operador.

4.8 Comentarios

Dart permite diferentes tipos de comentarios, entre los que se encuentran: de una sola línea, de varias líneas o comentarios de la documentación (*Effective DART: Documentation*, s.f.).

4.8.1 Comentarios de una sola línea

Todo aquello que se encuentre entre dos barras: *//*, el compilador de Dart lo omite. Por ende, en la salida solo se ejecuta las líneas que no tengan comentarios.

Así como se puede observar en el siguiente ejemplo:

```dart
void main() {
  //Mensaje de saludo
  print("Hola Uniamazonia");
}
```

```
Hola Uniamazonia
```

4.8.2 Comentarios de varias líneas

Para agregar comentarios a varias líneas se puede hacer uso de la barra y el asterisco para abrir indicando el inicio del comentario: */** e invertido con asterisco y barra: ***/**, para indicar el final del bloque comentado.

Tal es el caso del siguiente ejemplo:

```dart
void main() {
  /* Mensaje de saludo
  se utiliza un print para la salida
  del mensaje en consola.
  */
  print("Hola Uniamazonia");
}
```

```
Hola Uniamazonia
```

4.8.3 Comentarios de la documentación

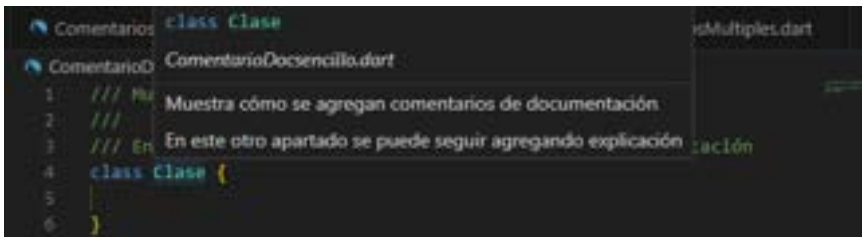
La documentación no solo está dirigida a los consumidores externos, sino que también resulta útil para los miembros internos que desean comprender el código desde diferentes perspectivas. Es importante tener en cuenta que la descripción del comentario debe estar centrada en el usuario. Se comienza con mayúscula, a menos que se haga referencia a un identificador sensible a mayúsculas y minúsculas. Además, es preferible terminar con un punto como buena práctica.

En este sentido, se debe recordar que los comentarios de documentación pueden ser de una o varias líneas. Se inicia el comentario con tres barras: ///, y cada línea que se desea comentar debe seguir este formato.

Como se puede observar en el siguiente ejemplo, se ilustra el funcionamiento básico de los comentarios de documentación.

```dart
/// Muestra cómo se agregan comentarios de documentación.
///
/// En este otro apartado se puede seguir agregando explicación.
class Clase {

}
```

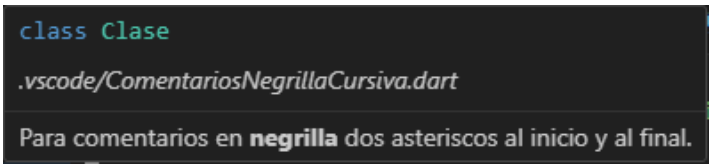
Para acceder a la documentación, se puede hacer fácilmente desde un editor de código fuente, como en este caso **Visual Studio Code**. Basta con ubicar el cursor sobre las clases, variables, atributos, funciones u otros elementos documentados, y automáticamente se mostrará una ventana emergente con la información agregada.



De la misma forma, se pueden agregar comentarios con estilo en formato Markdown para darle formato a la documentación. Por ejemplo, para poner un texto en negrita, se debe colocar entre dos asteriscos al inicio y al final del texto a formatear.

```dart
/// Para comentarios en **negrilla** dos asteriscos al inicio
/// y al final.
class Clase {

}
```

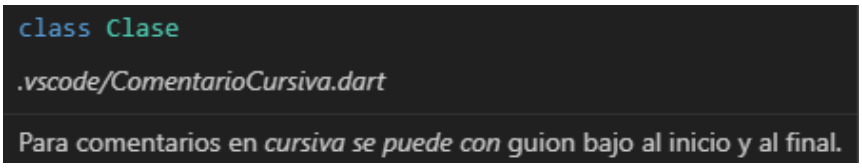
Quedando de la siguiente manera:



Asimismo, podemos hacer uso del guion bajo, también denominado "raya al piso" o subrayado, al inicio y al final del texto que se desea escribir en formato de cursiva.

```dart
/// Para comentarios en _cursiva se puede con_ guion bajo al inicio
/// y al final.
class Clase {

}
```

Dando como resultado los siguiente;



Dentro de los comentarios de documentación, se pueden agregar referencias entre corchetes para mencionar clases, métodos, variables y parámetros.

A continuación, se muestra un ejemplo del uso de este tipo de referencia, que se combina con un comentario de una línea, demostrando que es posible mezclar diferentes tipos de comentarios.

```dart
/// Clase que representa a una persona con su edad.
class Persona {
  int _edad; // Atributo privado para almacenar la edad.

  /// Constructor que inicializa la edad de la persona.
  Persona(this._edad);

  /// Método para obtener la edad actual de la persona.
  int get edad => _edad;

  /// Método para incrementar la edad de la persona en [cantidad] años.
  ///
  /// Este método toma un parámetro [cantidad] que representa la cantidad
  /// de años en los que se incrementará la edad de la persona.
  void incrementarEdad(int cantidad) {
    _edad += cantidad;
  }

  /// Método para decrementar la edad de la persona en [cantidad] años.
  ///
  /// Este método toma un parámetro [cantidad] que representa la cantidad
  /// de años en los que se decrementará la edad de la persona.
  void decrementarEdad(int cantidad) {
    if (_edad - cantidad >= 0) {
      _edad -= cantidad;
    } else {
      print('La edad no puede ser negativa.');
    }
  }
}
```

Adicionalmente se puede dar formato y usar comodines para mejorar la estética de los comentarios, para ello se comienza mirando los títulos.

```dart
void main() {
  saludar();
}

/// # Titulo 1
/// ## Titulo 2
/// ### Titulo 3
/// #### Titulo 4
void saludar() {
  print("Hola Uniamazonia + Comentarios");
}
```

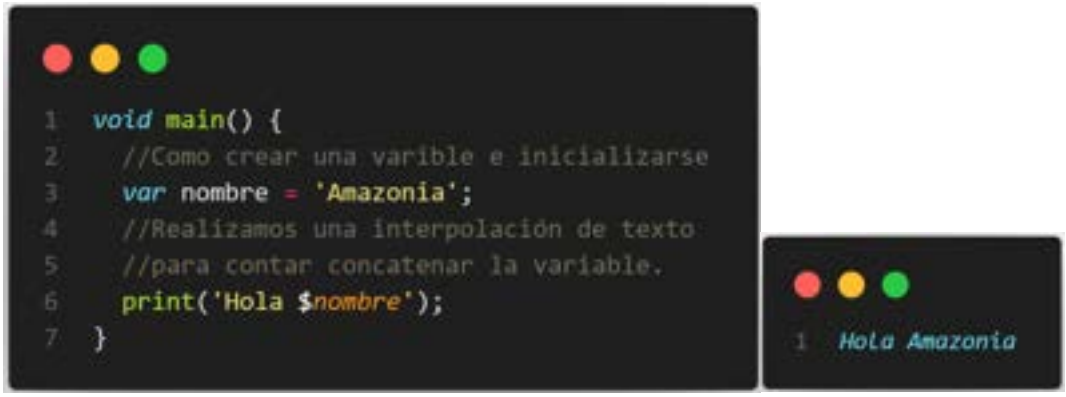
Utilizando una combinación de tabulación para indicar niveles y comodines, como el símbolo de suma (+), resta (-) o asterisco (*), los cuales se pueden utilizar mezclados o de forma individual, es posible generar listas con viñetas.

Ahora bien, si desea más información puede buscar dentro de la documentación oficial **Dart Doc** (dart doc. s.f.) sobre cómo crear la referencia del código fuente de Dart.

4.9 Hola Amazonia

En Dart, las variables pueden declarar explícitamente su tipo de dato o no hacerlo. Cuando no se indica el tipo, el lenguaje infiere automáticamente el tipo según el valor asignado. Para omitir la declaración explícita, se puede utilizar la palabra clave **var**, como se muestra a continuación. Sin embargo, es recomendable especificar el tipo de dato para mejorar la claridad y mantenibilidad del código.

Además, con la función **print**, es posible imprimir valores en la consola.



4.10 Tipos de datos

Dart es un lenguaje de programación que cuenta con diversos tipos de datos (Variables, s. f.). Algunos de los más comunes son los siguientes; sin embargo, existen muchos otros, como los tipos de datos para manejar fechas y horas, así como los tipos de datos para expresiones regulares.

4.10.1 numéricos

Los **int** son tipos de dato que se utilizan para almacenar números enteros. Por ejemplo, 10, -20, 0.

Así como los **double** se utilizan para almacenar números con decimales. Por ejemplo, 3.1416, -1.2,

```dart
void main() {
  //numero
  int empleado = 10;
  double altura = 1.87;
  // Salida en consola
  print('Entero:$empleado, Altura:$altura');
}
```

```
Entero:10, Altura:1.87
```

4.10.2 String

Este tipo de dato es utilizado para almacenar cadenas de texto. Por ejemplo, "Amazonia", 'Hola mundo', '2024', se puede utilizar comillas dobles o sencillas.

```dart
void main() {
  String nombre = 'Desde la Amazonia';
  print(nombre);
  print(nombre[0]);
  print(nombre[9]);
  print(nombre[nombre.length - 2]);
}
```

```
Desde la Amazonia
D
A
i
```

4.10.3 Conversión de tipos de datos (Numérico - Texto)

El objetivo es convertir variables de tipo **String** a valores numéricos y viceversa. En este sentido, si se tiene un número almacenado en una variable de tipo texto, se puede aplicar sobre esta el método **parse**(). Es importante tener en cuenta que este método puede utilizarse tanto con números enteros como con decimales, de la siguiente manera:

```dart
void main() {
  String medida = '80';
  int edad = int.parse(medida);
  print('La edad es $edad');
}
```

```
La edad es 80
```

Del mismo modo podemos trabajar para convertir a decimales.

```dart
void main() {
  String medida = '80.9';
  double peso = double.parse(medida);
  print('El peso $peso');
}
```

```
El peso 80.9
```

En ocasiones, no se tiene claro el tipo de dato resultante y, por ejemplo, se puede intentar almacenar un número con decimales dentro de una variable entera. Sin embargo, durante el proceso de conversión, esto puede generar una **Exception**.

```dart
void main() {
  String medida = '80.9';
  int peso = int.parse(medida);
  print('El peso $peso');
}
```

```
Se produjo una excepción. ×
FormatException (FormatException: Invalid
radix-10 number (at character 1)
80.9
^
)
```

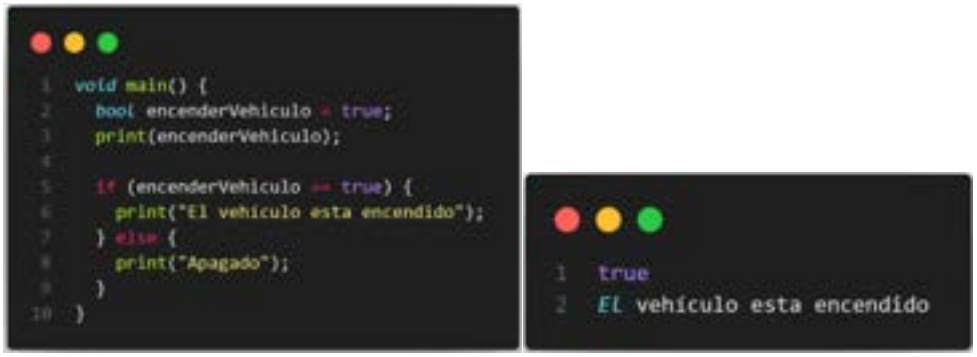
Por último, para convertir una variable de tipo numérico a **String**, se utiliza el método **toString**(), el cual está disponible en la mayoría de las variables.

```dart
void main() {
  int medida = 2025;
  String modelo = medida.toString();
  print(modelo);
}
```

```
2025
```

4.10.4 Booleano

Este tipo de dato se utiliza para almacenar valores **true** o **false**.



```dart
void main() {
  bool encenderVehiculo = true;
  print(encenderVehiculo);

  if (encenderVehiculo == true) {
    print("El vehiculo esta encendido");
  } else {
    print("Apagado");
  }
}
```

```
true
El vehiculo esta encendido
```

Se puede quitar el **== true** por la naturaleza del **if** y el valor de la variable.

```dart
void main() {
  bool encenderVehiculo = true;
  print(encenderVehiculo);

  if (encenderVehiculo) {
    print("El vehiculo esta encendido");
  } else {
    print("Apagado");
  }
}
```

```
true
El vehiculo esta encendido
```

Es posible negar un valor booleano anteponiendo ! a la variable, lo que invierte su valor lógico.

```dart
void main() {
  bool encenderVehiculo = true;
  print(encenderVehiculo);
  encenderVehiculo = !encenderVehiculo;
  print(encenderVehiculo);
  if (encenderVehiculo == true) {
    print("El vehiculo esta encendido");
  } else {
    print("Apagado");
  }
}
```

```
true
false
Apagado
```

La negación también se puede aplicar directamente en la evaluación.

```dart
void main() {
  bool encenderVehiculo = true;
  print(encenderVehiculo);

  if (!encenderVehiculo) {
    print("El vehiculo esta encendido");
  } else {
    print("Apagado");
  }
}
```

```
true
Apagado
```

4.10.5 Lista

Este tipo de dato se utiliza para almacenar una colección de valores. Los elementos de una lista pueden ser de cualquier tipo de dato, pero en conjunto del mismo tipo. Por ejemplo, [‘a’, ‘b’, ‘c’], [1, 2, 3] o [true, false, false],

De esta forma puede trabajar con una lista de elementos. (es como usar una lista de tipo de dato dinámico)

```dart
void main() {
  List vocales = ['a', 'e', 'i', 'o', 'u'];
  print(vocales);
}
```

```
[a, e, i, o, u]
```

El método **add** permite agregar elementos de tipo dinámico o el defino al momento de crearla.

```dart
void main() {
  List numeros = [1, 2, 3, 4, 5];
  print(numeros);
  numeros.add(6);
  print(numeros);
}
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
```

Además, permite agregar texto en lugares donde originalmente se almacenaban números, debido a que no tiene un tipo de dato definido.

```dart
void main() {
  List numeros = [1, 2, 3, 4, 5];
  print(numeros);
  numeros.add(6);
  print(numeros);
  numeros.add('siete');
  print(numeros);
}
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, siete]
```

Sin embargo, si se desea restringir el tipo de dato, este se indica entre los signos < >.

```dart
void main() {
  List<int> numeros = [1, 2, 3, 4, 5];
  print(numeros);
  numeros.add(6);
  print(numeros);
  numeros.add('siete');
  print(numeros);
}
```

```
                                    6:15: Error:
The argument type 'String' can't be ass
igned to the parameter type 'int'.
  numeros.add('siete');
              ^
```

El método nos indica el tipo de argumento que debe llevar.

Desde la versión 2.12, Dart ha incorporado la comprobación de seguridad **Sound Null Safety**, la cual impide que se asignen valores **null** a las variables, incluyendo las listas. Por esta razón, es necesario asignar valores a los elementos de la lista.

```dart
void main() {
  List numeros = new List(27);
  print(numeros);
}
```

```
List(27);
The class 'List' doesn't have an unnamed constructor.
Try using one of the named constructors defined in
'List'. dart(new_with_undefined_constructor_default)
```

Nota: También se pueden utilizar otras funciones propias de las listas, entre ellas:

Eliminar un elemento por su posición en la lista: **lista.removeAt**(indice);

```dart
void main(){
  List<String> municipios = ['Florencia','Morelia','El Doncello'];
  print(municipios);
  municipios.removeAt(0);
  print(municipios);
}
```

```
[Florencia, Morelia, El Doncello]
[Morelia, El Doncello]
```

Borrar el último elemento: **lista.removeLast**();

```dart
void main(){
  List<String> municipios = ['Florencia','Morelia','El Doncello'];
  print(municipios);
  municipios.removeLast();
  print(municipios);
}
```

```
[Florencia, Morelia, El Doncello]
[Florencia, Morelia]
```

Añadir elementos de otra lista: **lista.addAll(nuevaLista);**

```dart
void main(){
  List<String> aves = ['Guacamayo','Tucán'];
  print(aves);
  List<String> nuevasAves = ['Colibrí'];
  print(nuevasAves);
  aves.addAll(nuevasAves);
  print(aves);
}
```

```
[Guacamayo, Tucán]
[Colibrí]
[Guacamayo, Tucán, Colibrí]
```

4.10.6 Set

Este tipo de dato es utilizado para almacenar un conjunto de valores únicos. Los elementos de un set pueden ser de cualquier tipo de dato. Por ejemplo, {27, 3, 21}, {‘a’,’e’,’i’}, {true, false},

```dart
void main(){
  Set <String> animales ={};
  animales ={'Águila','Tucán'};
  print(animales);
}
```

```
{Águila, Tucán}
```

Agregar un elemento utilizando el método **add**.

```dart
void main(){
  Set <String> animales ={};
  animales ={'Águila','Tucán'};
  print(animales);
  animales.add('Colibrí');
  print(animales);
}
```

```
{Águila, Tucán}
{Águila, Tucán, Colibrí}
```

Si se intenta agregar un valor ya existente, este no surtirá efecto dentro del conjunto de valores. Es importante tener en cuenta que, dependiendo del editor utilizado, el conjunto puede permitir valores duplicados o simplemente indicar que el valor ya existe.

```dart
void main(){
  Set <String> animales ={};
  animales ={'Águila','Tucán'};
  print(animales);
  animales.add('Colibrí');
  print(animales);
  animales.add('Tucán');
  print(animales);
}
```

```
{Águila, Tucán}
{Águila, Tucán, Colibrí}
{Águila, Tucán, Colibrí}
```

Para recorrer los valores dentro del conjunto, se utiliza el método **elementAt**().

```dart
void main(){
  Set <String> animales = {};
  animales = {'Águila','Tucán','Colibrí'};

 for(int i = 0; i < animales.length ; i++ ){
  print(animales.elementAt(i));
 }
}
```

```
Águila
Tucán
Colibrí
```

Nota: Tener en cuenta que existen otra gran cantidad de métodos bastantes útiles que se pueden explorar.

4.10.7 Map

Se utiliza para almacenar un conjunto de pares compuestos por clave-valor. Las claves y los valores de un mapa pueden ser de cualquier tipo de dato. Por ejemplo, {'a': 1, 'b': 2, 'c': 3}, {1: 'a', 2: 'b', 3: 'c'}, {true: false, false: true}.

Son pares de valores el primero es la llave y el segundo el valor. Ahora bien, según esta estructura tanto la llave como el valor son dinámicos porque no definimos el tipo.

```dart
void main(){
  Map animal = {
    'Género':'Callicebus',
    'Especie':'Callicebus caquetensis',
    'Tamaño':35
  };
}
```

Sin embargo, es posible definir el tipo de la llave y del valor. En este caso, se especifica que la llave debe ser de tipo String, mientras que el valor puede ser de cualquier tipo.

```dart
void main(){
  Map<String,dynamic> animal = {
    'Género':'Callicebus',
    'Especie':'Callicebus caquetensis',
    'Tamaño':35,
  };
}
```

Ejemplo con los dos dinámicos para comprobar lo dicho anteriormente.

```dart
void main(){
  Map<dynamic,dynamic> animal = {
    'Género':'Callicebus',
    'Especie':'Callicebus caquetensis',
    'Tamaño':35,
    1:"En peligro crítico"
  };
}
```

Para la visualización de los valores del mapa, es necesario considerar que la llave se utiliza dentro de los corchetes [ ]. En el caso de intentar acceder mediante un punto, tal como en otros lenguajes, se aludirá a los atributos inherentes al Map.

```dart
void main(){
  Map<dynamic,dynamic> animal = {
    'Género':'Callicebus',
    'Especie':'Callicebus caquetensis',
    'Tamaño':35,
    1:"En peligro crítico"
  };
  print(animal['Género']);
  print(animal[1]);
  print(animal);
}
```

```
Callicebus
En peligro crítico
{Género: Callicebus, Especie: Callicebus caquetensis, Tamaño: 35, 1: En peligro crítico}
```

Es posible guardar la clave dentro de una variable para luego llamarla en ejecución.

```dart
void main(){
  String especie = 'Especie';
  Map<dynamic,dynamic> animal = {
    'Género':'Callicebus',
    'Especie':'Callicebus caquetensis',
    'Tamaño':35,
    1:"En peligro crítico"
  };
  print(animal['Género']);
  print(animal[especie]);
}
```

```
Callicebus
Callicebus caquetensis
```

Una de las ventajas del tipado de datos es que permite definir qué valores puede recibir un **Map**. Por ejemplo, se puede especificar que la llave sea un número y el valor un **String**, restringiendo automáticamente los tipos de datos permitidos, tal como se observa en la documentación.

```dart
void main(){
  Map<int, String> estadoConservacion ={
   1:'Extinto',
   2:'Amenazado'
  };

  estadoConservacion.addAll()
}
```

Ahora bien, si se intenta ingresar un conjunto de pares que no cumpla con los tipos de datos definidos, se generará un error.

```dart

void main(){
  Map<int, String> estadoConservacion ={
   1:'Extinto',
   2:'Amenazado'
  };

  estadoConservacion.addAll({'3':'Preocupacion'});
}
```

Sin embargo, si se cumple con el tipo definido, no habrá ningún problema. Además, basta con utilizar el nombre del **Map** para visualizar todos los objetos que contiene.

```dart
void main(){
    Map<int, String> estadoConservacion ={
     1:'Extinto',
     2:'Amenazado'
    };

    estadoConservacion.addAll({3:'Preocupación'});
    print(estadoConservacion);
  }
```

```
{1: Extinto, 2: Amenazado, 3: Preocupación}
```

Ahora bien, si se desea visualizar un elemento específico, se debe indicar el nombre del **Map** y, entre corchetes, el valor de la llave. Es importante destacar que no se utiliza la posición, ya que un **Map** no es un arreglo, sino un diccionario de datos.

```dart
void main(){
    Map<int, String> estadoConservacion ={
     1:'Extinto',
     2:'Amenazado'
    };

    estadoConservacion.addAll({3:'Preocupación'});
    print(estadoConservacion[3]);
  }
```

```
Preocupación
```

4.10.8 Enumeraciones

Las enumeraciones, o **enum**, permiten manejar opciones limitadas dentro de un conjunto predefinido de valores. Para acceder a un valor dentro de un **enum**, es necesario utilizar el nombre de la enumeración seguido del nombre del valor correspondiente.

```dart
void main(){
    gruposIndigenas indigenas = gruposIndigenas.Uitotos;

    switch(indigenas){
      case gruposIndigenas.Coreguages:
        print('Los Coreguages');
        break;
      case gruposIndigenas.Inganos:
        print('Los Inganos');
         break;
      case gruposIndigenas.Uitotos:
        print('Los Uitotos');
        break;
    }
  }

  enum gruposIndigenas{Uitotos, Coreguages, Inganos}
```

```
Los Uitotos
```

4.11 Operaciones

4.11.1 Aritméticas

Dart como muchos otros lenguajes manejan los operadores aritméticos tradicionales como lo son:

Operador	Descripción
+	Agregar
-	Sustraer
*	Multiplicar
/	Dividir
~/	Dividir devolviendo solo la parte entera
%	Residuo

Se ilustra el funcionamiento de los operadores aritméticos.

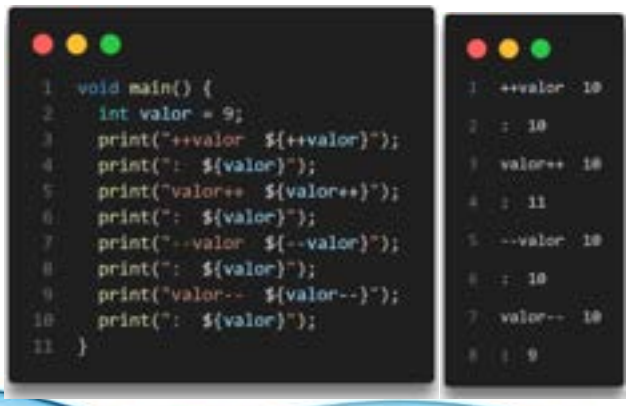
```
void main() {
  print(2 + 9);
  print(2 - 9);
  print(2 * 9);
  print(2 / 9);
  print(9 ~/ 2);
  print(9 % 2);
}
```

```
11
-7
18
0.2222222222222222
4
1
```

Dentro de los aritméticos podemos incluir los unarios para incremento o decremento.

Operador	Descripción
++variable	Es el valor de la variable + 1
variable++	Es el valor de la variable. (después incrementa)
--variable	Es el valor de la variable – 1
variable--	Es el valor de la variable (después decrementa)

Se ilustra el funcionamiento de los operadores unitarios.



```
void main() {
  int valor = 9;
  print("++valor  ${++valor}");
  print(":  ${valor}");
  print("valor++  ${valor++}");
  print(":  ${valor}");
  print("--valor  ${--valor}");
  print(":  ${valor}");
  print("valor--  ${valor--}");
  print(":  ${valor}");
}
```

```
++valor  10
:  10
valor++  10
:  11
--valor  10
:  10
valor--  10
:  9
```

4.11.2 Igualdad y relacionales

A continuación, se evidencian los operadores tanto de igualdad como relacionales.

Operador	Descripción
==	Igual a
!=	No es igual a
>	Mayor a
<	Menor a
>=	Mayor igual a
<=	Menor igual a

Se ilustra el comportamiento de los diferentes operadores tanto de igualdad como relacional.

```dart
void main() {
  int variable = 9;
  print(variable == 9);
  print(variable != 27);
  print(variable > 27);
  print(variable < 27);
  print(variable >= 21);
  print(variable <= 4);
}
```

```
true
true
false
true
false
false
```

4.11.3 Lógicos

Operador	Descripción
! expresión	Invierte el valor de verdadero a falso y viceversa
||	El O lógico
&&	El Y lógico

Se ilustra el comportamiento de los diferentes operadores.

```dart
void main() {
  bool variable = true;
  print(!variable);

  if (3 < 21 || 3 > 21) {
    print("Se cumplio alguna de las proposición");
  }

  if (3 < 21 && 8 > 3) {
    print("Se cumplieron todas las proposición");
  }
}
```

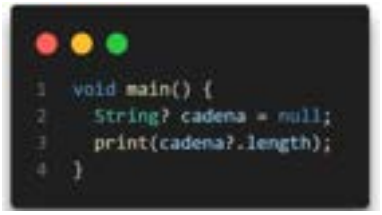
```
false
Se cumplio alguna de las condiciones
Se cumplieron todas las condiciones
```

4.11.4 Operadores Null

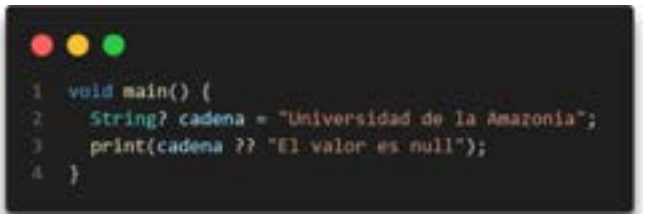
Operador	Descripción
?	Un valor potencialmente null
??	Si la expresión es null devuelve la de la derecha de lo contrario la de la izquierda.
?? =	Valida si se encuentra en null; de ser así asigna el valor a la variable de lo contrario continua con el valor.

A continuación, ilustraciones de la aplicación de los operadores:

También es posible agregar un signo de interrogación (?) antes del punto para manejar valores nulos. Esto se debe a que, si se envía un valor null, Dart detecta la posibilidad de un null y genera un error debido a la funcionalidad **Sound Null Safety**.



Comportamiento del operador (**??**) cuando el valor no es null



El mismo operador, pero con valor null.

Mediante el operador (?=), se realiza una validación y, en caso de que la variable sea null, se asigna el valor correspondiente.

```dart
void main() {
  String? cadena;
  cadena ??= "Universidad de la Amazonia";
  print(cadena);
}
```

```
Universidad de la Amazonia
```

En caso de que no llegue a la validación del operador un null, deja el valor que viene.

```dart
void main() {
  String? cadena = "Uniamazonia";
  cadena ??= "Universidad de la Amazonia";
  print(cadena);
}
```

```
Uniamazonia
```

4.11.5 Operadores de prueba de tipo

Son básicos para comprobar el tipo de datos en tiempo de ejecución.

Operador	Descripción
as	Typecast (Convertir un objeto a un tipo particular)
is	Si el tipo del objeto es el mismo será verdadero
is!	Si el tipo del objeto no es el mismo será verdadero

El operador **is** permite comparar los tipos de datos, de manera similar al operador ==, utilizado para verificar la igualdad.

```dart
void main() {
  if ("Uniamazonia" is String) {
    print("Si es");
  } else {
    print("No es");
  }
}
```

```
Si es
```

De manera similar, se puede realizar la comparación para verificar si un valor es diferente, utilizando el operador is!.

```dart
void main() {
  if ("Uniamazonia" is! String) {
    print("Si es");
  } else {
    print("No es");
  }
}
```

```
No es
```

4.12 Funciones

Las funciones tienen una estructura compuesta por el valor de retorno, el nombre de la función y, dentro de paréntesis, los parámetros (*Functions*, s.f.).

T

```dart
void main() {

}
```

Al definir una función en Dart sin especificar explícitamente el valor de retorno, el lenguaje infiere su tipo como **Dynamic**, lo que indica que la función puede devolver cualquier tipo de dato. Si bien esta flexibilidad no es incorrecta, es recomendable definir el tipo de retorno de manera explícita para garantizar un código más estructurado, facilitar su mantenimiento y mejorar la robustez del software.

Cuando la función **saludar** se incorpora dentro de **main**, esta se ejecuta y su resultado se muestra en pantalla, ya que pasa a formar parte del flujo de ejecución del programa y es invocada en tiempo de ejecución.

```dart
void main() {
  saludar();
}

saludar() {
  print("Hola Uniamazonia");
}
```

```
Hola Uniamazonia
```

Es posible indicar que una función no retorne ningún valor utilizando la palabra reservada **void**. Es importante tener en cuenta que la instrucción **print** no representa un valor de retorno de la función, sino simplemente una línea de código que imprime un mensaje en pantalla.

```dart
void main() {
  saludar();
}

void saludar() {
  print("Hola Uniamazonia con el void");
}
```

```
Hola Uniamazonia con el void
```

Si se desea retornar un valor, como un **String**, es posible indicarlo al inicio de la función, tal como se observa en la imagen. No obstante, en caso de que la función no incluya una instrucción de retorno, se generará un error indicando que no se está devolviendo ningún valor.

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

String saludo(){
  print('hola desde la Amazonia');
}
```

```
Error: A non-null value must be returned since the return type 'String' doesn't allow null.
```

Para solucionar este problema, es necesario utilizar la palabra reservada **return** dentro de la función **saludar**. Sin embargo, este no es el único ajuste requerido para visualizar el resultado en pantalla. Dentro del **main**, donde se llama a la función, se debe emplear la función **print** para imprimir el valor retornado. Es importante recordar que ahora la función **saludar** retorna un **String**, por lo que su salida deberá tratarse como tal.

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

String saludo(){
  return ('hola desde la Amazonia');
}
```

```
hola desde la Amazonia
```

Otra opción consiste en almacenar el valor retornado por la función dentro de una variable de tipo **String**, dado que ambos tipos de datos son compatibles. Posteriormente, se puede utilizar la función **print** para mostrar el contenido de dicha variable en pantalla.

```dart
void main(){
  String mensaje = saludo();
  print(mensaje);
}

String saludo(){
  return ('hola desde la Amazonia');
}
```

```
hola desde la Amazonia
```

4.12.1 Parámetros posicionales

Como se indicó en la estructura de la función, es posible enviar argumentos colocando las variables dentro de los paréntesis y utilizándolas dentro de la lógica de esta.

Al ejecutarla, el programa funciona correctamente, ya que toma el valor proporcionado como argumento dentro de la función **saludar** en el **main**. Sin embargo, no se están especificando los tipos de datos para los parámetros de la función, lo que hace que Dart los detecte como **Dynamic**, permitiendo el envío de cualquier tipo de dato.

```dart
void main() {
  String mensaje = saludar('Oscar', 'Buenos dias');
  print(mensaje);
}

String saludar(persona, texto) {
  return 'Hola $persona $texto';
}
```

```
Hola Oscar Buenos dias
```

Para hacer que los parámetros sean estrictos, es necesario indicar explícitamente sus tipos de datos. Esto permite que Dart valide y garantice que los tipos de los argumentos coincidan con los parámetros definidos.

Por ejemplo, si una función recibe un primer parámetro de tipo **String** y otro de tipo **int**, y se intenta pasar un valor de un tipo diferente, el compilador generará un error.

```dart
void main() {
  String mensaje = saludar('Oscar', 'Buenos dias');
  print(mensaje);
}

String saludar(String persona, int edad) {
  return 'Hola $persona $edad';
}
```

```
parametrosConTipo.dart:2:37: Error: The ar…o.dart:2
gument type 'String' can't be assigned to
the parameter type 'int'.
  String mensaje = saludar('Oscar', 'Buenos dias');
                                    ^
```

Sin embargo, al realizar el ajuste e ingresar valores que coincidan con los tipos definidos en los parámetros, la función se ejecuta correctamente sin generar errores.

```dart
void main() {
  String mensaje = saludar('Oscar', 20);
  print(mensaje);
}

String saludar(String persona, int edad) {
  return 'Hola $persona $edad';
}
```

```
Hola Oscar 20
```

4.12.2 Parámetros por nombre

El parámetro por nombre permite especificar los argumentos en cualquier orden, indicando explícitamente el nombre del parámetro, como se evidencia a continuación. Sin embargo, es importante considerar que, al definir la función, los parámetros deben estar envueltos entre llaves `{}`. Además, debido a la naturaleza del manejo de valores **null** en Dart, es necesario controlar esta situación, lo cual se puede lograr de tres maneras.

```dart
void main() {
  String mensaje = saludar('Oscar', 20);
  print(mensaje);
}

String saludar({String persona, int edad}) {
  return 'Hola $persona  tu edad es $edad años';
}
```

```
parametros_nombre_v1.dart:6:24: Error: The parameter 'persona' can't have a value of 'null' because of its type 'String', but the implicit default value is 'null'.
Try adding either an explicit non-'null' default value or the 'required' modifier.
String saludar({String persona, int edad}) {
                       ^^^^^^^

parametros_nombre_v1.dart:6:37: Error: The parameter 'edad' can't have a value of 'null' because of its type 'int', but the implicit default value is 'null'.
Try adding either an explicit non-'null' default value or the 'required' modifier.
String saludar({String persona, int edad}) {
                                    ^^^^
```

La primera forma de manejar valores null en parámetros por nombre es asignar un valor por defecto. Como se muestra a continuación, esta práctica permite corregir el error en la función al garantizar que siempre haya un valor asignado al parámetro, incluso si no se proporciona un argumento al momento de la llamada.

```dart
void main() {
  String mensaje = saludar('Oscar', 20);
  print(mensaje);
}

String saludar({String persona = '', int edad = 0}) {
  return 'Hola $persona  tu edad es $edad años';
}
```

```
parametros_nombre_v1.dart:2:27: Error: Too many positional arguments: 0 allowed, but 2 found.
Try removing the extra positional arguments.
  String mensaje = saludar('Oscar', 20);
                           ^^^^^^^

parametros_nombre_v1.dart:6:8: Context: Found this candidate, but the arguments don't match.
String saludar({String persona = '', int edad = 0}) {
       ^^^^^^^
```

La segunda opción es utilizar el operador **?** después del tipo de dato para indicar que el parámetro puede aceptar valores **null**. Esto permite controlar y manejar adecuadamente la verificación de nulabilidad impuesta por el **Sound Null Safety** en Dart.

```dart
void main() {
  String mensaje = saludar('Oscar', 20);
  print(mensaje);
}

String saludar({String? persona, int? edad}) {
  return 'Hola $persona  tu edad es $edad años';
}
```

```
parametros_nombre_v1.dart:2:27: Error: Too many po
sitional arguments: 0 allowed, but 2 found.
Try removing the extra positional arguments.
  String mensaje = saludar('Oscar', 20);
                           ^
parametros_nombre_v1.dart:6:8: Context: Found this
candidate, but the arguments don't match.
String saludar({String? persona, int? edad}) {
```

Ahora bien, una vez controlada la nulabilidad en Dart, se pueden utilizar cualquiera de las dos formas mencionadas para gestionar los valores **null**. Con esto, es posible manejar el envío de argumentos mediante parámetros con nombre.

Asimismo de parámetros con nombre permite especificar los valores de los argumentos en cualquier orden al momento de invocar la función. Para ello, se indica el nombre del parámetro seguido del valor correspondiente. De esta manera, la función reconocerá y asignará correctamente cada argumento a su respectivo parámetro sin generar errores por el orden en que se envían.

```dart
void main() {
  String mensaje = saludar(edad: 20, persona: 'Oscar');
  print(mensaje);
}

String saludar({String? persona, int? edad}) {
  return 'Hola $persona  tu edad es $edad años';
}
```

```
Hola Oscar  tu edad es 20 años
```

Además, los argumentos pueden proporcionarse en cualquier orden, ya que Dart asocia cada valor con el parámetro correspondiente mediante su nombre. Esto permite mayor flexibilidad al momento de invocar la función sin preocuparse por el orden estricto de los parámetros.

```dart
void main() {
  String mensaje = saludar(persona: 'Oscar', edad: 20);
  print(mensaje);
}

String saludar({String? persona, int? edad}) {
  return 'Hola $persona  tu edad es $edad años';
}
```

```
Hola Oscar  tu edad es 20 años
```

La tercera y última opción consiste en utilizar la palabra reservada **required**, la cual indica a Dart que el parámetro es obligatorio al momento de invocar la función. De este modo, se evita que el valor sea **null** y se garantiza que la función reciba los datos necesarios para su ejecución.

```dart
void main() {
  String mensaje = saludar(persona: 'Oscar', edad: 20);
  print(mensaje);
}

String saludar({required String persona, required int edad}) {
  return 'Hola $persona  tu edad es $edad años';
}
```

```
Hola Oscar  tu edad es 20 años
```

4.12.3 Argumentos opcionales

Es posible agregar parámetros opcionales, los cuales permiten que los argumentos sean enviados o no. Para ello, se encierra el parámetro entre corchetes **[ ]** y se le asigna un valor por defecto. De esta manera, si no se proporciona un argumento al invocar la función, se utilizará el valor definido por defecto.

```dart
void main() {
  String mensaje = saludar();
  print(mensaje);

  String mensajeNuevo = 'Hola mis amigos !!!';
  print(saludar(mensajeNuevo));
}

String saludar([String saludo = 'Hola mi gente !!!']) {
  return '$saludo';
}
```

```
Hola mi gente !!!
Hola mis amigos !!!
```

4.12.4 Funciones flecha (Arrow Function)

Son funciones diseñadas para ejecutarse en una sola línea de código, lo que permite simplificar y mejorar la legibilidad del código. En lugar de utilizar llaves **{ }**, se reemplazan por la denominada **flecha (=>)**. Además, es importante tener en cuenta que la palabra reservada **return** no es necesaria, ya que la función devuelve automáticamente el valor resultante del código especificado. Cabe destacar que el tipo de dato retornado coincide con el tipo definido en la declaración de la función.

3

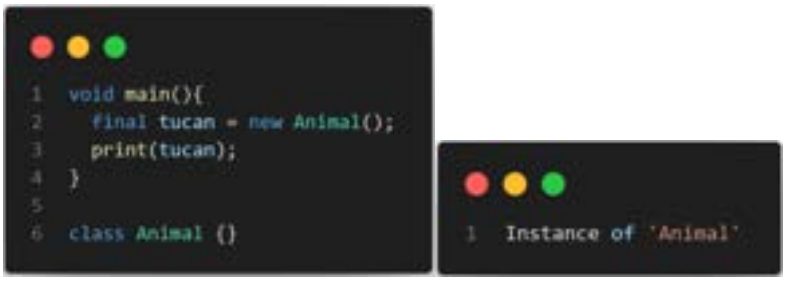
```dart
void main() {
  String mensaje = saludar();
  print(mensaje);

  String mensajeNuevo = 'Hola mis amigos !!!';
  print(saludar(mensajeNuevo));
}

String saludar([String saludo = 'Hola mi gente !!!']) => '$saludo';
```

```
Hola mi gente !!!
Hola mis amigos !!!
```

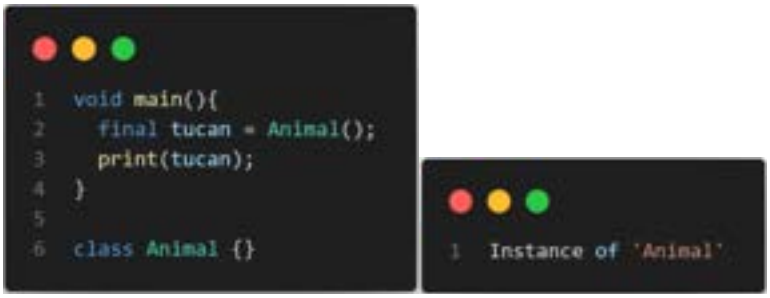
5 CLASES

Como se sabe, en programación una **clase** es una plantilla que permite crear **objetos**.

Uno de los cambios introducidos en versiones recientes de **Dart** es que, al momento de instanciar una clase, anteriormente se utilizaba la palabra reservada **new**. Sin embargo, en las versiones más recientes, el uso de **new** es opcional, debido a que Dart permite instanciar objetos sin necesidad de esta palabra clave, haciendo el código más limpio y conciso (*Classes*, s.f.).



Sin embargo, actualmente **Dart** infiere automáticamente que se está instanciando un objeto, por lo que no es necesario escribir explícitamente la palabra reservada **new**. Aun así, por buenas prácticas, se recomienda su uso, aunque omitirlo no generará ningún error en la ejecución del programa.



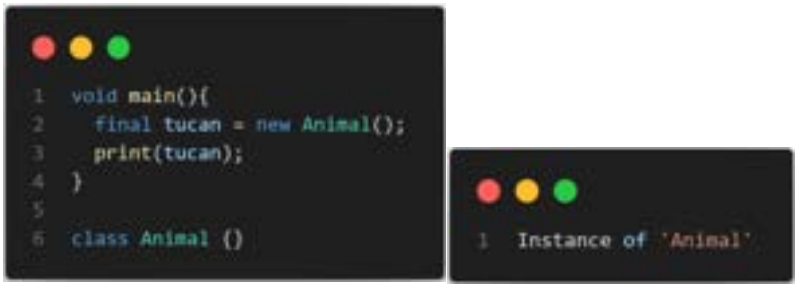
En **Dart**, la palabra reservada **final** se utiliza para declarar variables y objetos cuyo valor no puede ser reasignado después de su inicialización. Esto significa que, una vez asignado un valor a una variable o un objeto declarado como **final**, este no puede ser modificado.

Sin embargo, es importante tener en cuenta que **final** no hace que los objetos sean completamente inmutables, dado que, si la variable contiene una referencia a un objeto, sus propiedades aún pueden modificarse, a menos que sean declaradas como **final** o **const** dentro de la clase.

En el caso de las clases, el modificador **final** impide que sean extendidas o subclaseadas.

```
final valor = [1,2,3,4];
print(valor); //imprime el array [1,2,3,4], valor asignado inicialmente.
valor.add(5); //Se hace uso del método para agregar valor al array
print(valor); //imprime el array [1,2,3,4,5], Observamos que el valor inicial del array
```

Teniendo en cuenta lo anterior, una clase se puede instanciar de la siguiente manera. Incluso, si se imprime en consola el valor de la variable que almacena la instancia, se puede observar que esta ha sido correctamente creada y está en uso.

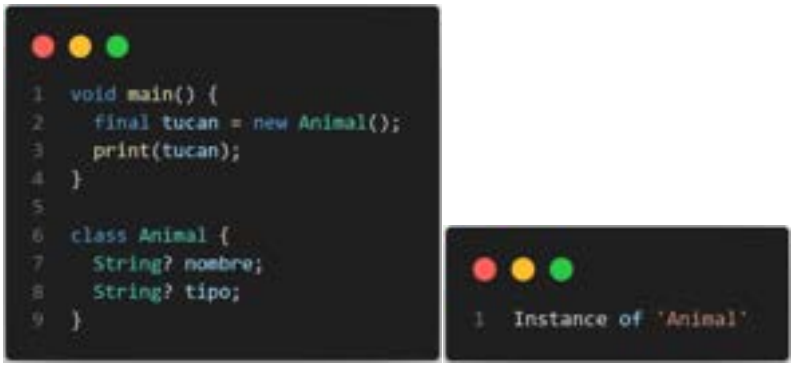


O incluso, se puede indicar explícitamente el tipo de dato del objeto, declarando la variable (que posteriormente será el objeto) con el nombre de la clase, especificando así su tipo. Esto se realiza de la siguiente manera:

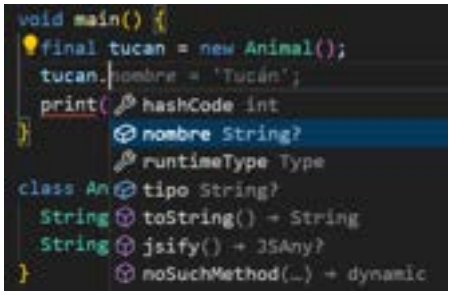
Cualquiera de las formas indicadas anteriormente nos permite instanciar a Animal.

5.1 Atributos o propiedades de las clases

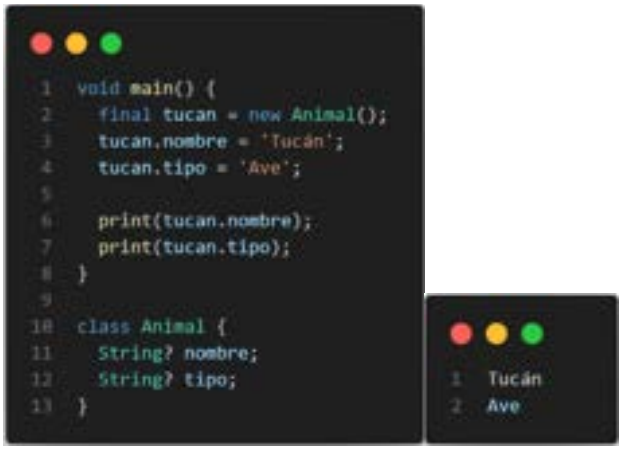
Se le pueden agregar atributos o propiedades a la clase indicando el tipo y el nombre de una variable. En este caso, se ha utilizado el operador ? para permitir que el atributo pueda contener valores nulos, como parte del control de *null safety* en Dart.



Ahora bien, al acceder a la variable que representa el objeto, se puede notar que al llamarla se muestran algunos atributos con los que se puede interactuar. Entre estos se encuentran las propiedades de la clase que fueron definidas previamente.



Se pueden asignar valores a las propiedades del objeto utilizando el operador =. Para ello, basta con indicar el objeto, seguido de la propiedad que se desea modificar. Posteriormente, se puede visualizar el valor asignado utilizando un método como **print**, que mostrará en consola el contenido de la propiedad.



5.2 Constructores

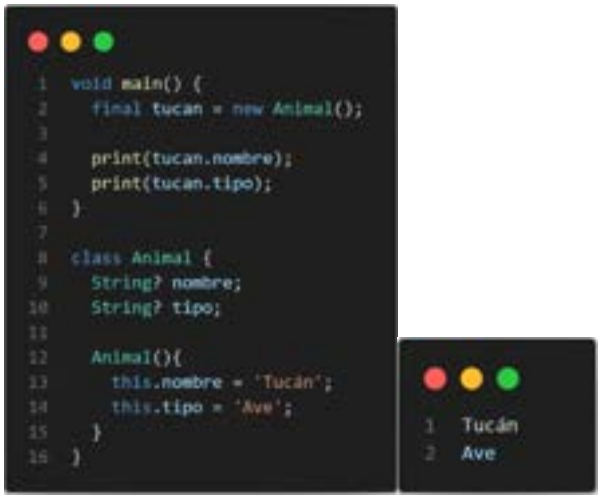
Sin embargo, es posible crear constructores para inicializar los valores de las propiedades de la clase. En este caso, se define un constructor utilizando el mismo nombre de la clase y especificando las propiedades como parámetros.

Nota: En Dart, existen tres tipos de constructores:

1. **Por defecto**: Es aquel que no tiene parámetros y tiene el mismo nombre de la clase.
2. **Con parámetros**: Permite enviar argumentos al momento de instanciar la clase. Sin embargo, no se puede tener simultáneamente un constructor por defecto y uno con parámetros dentro de la misma clase.
3. **Nombrado**: Es aquel al que se le asigna un nombre personalizado. Pueden coexistir varios constructores nombrados dentro de la misma clase.

5.2.1 Constructor por defecto

El constructor por defecto ya está presente al momento de crear una clase. Para definirlo explícitamente, se crea un método con el mismo nombre de la clase, sin parámetros y con sus respectivas llaves { }.

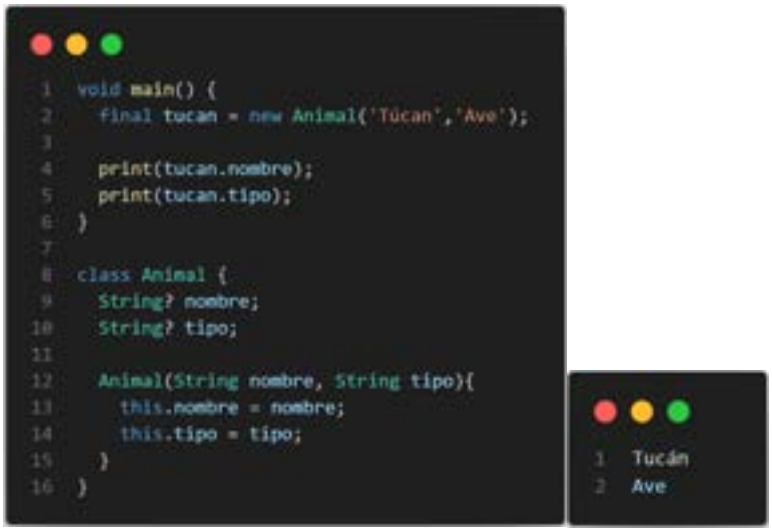


```dart
void main() {
  final tucan = new Animal();

  print(tucan.nombre);
  print(tucan.tipo);
}

class Animal {
  String? nombre;
  String? tipo;

  Animal(){
    this.nombre = 'Tucán';
    this.tipo = 'Ave';
  }
}
```

```
Tucán
Ave
```

5.2.2 Constructor por parámetro

Para diferenciar entre la variable del parámetro y las propiedades de la clase cuando tienen el mismo nombre, se utiliza la palabra reservada this, anteponiéndola a la propiedad como diferenciador.

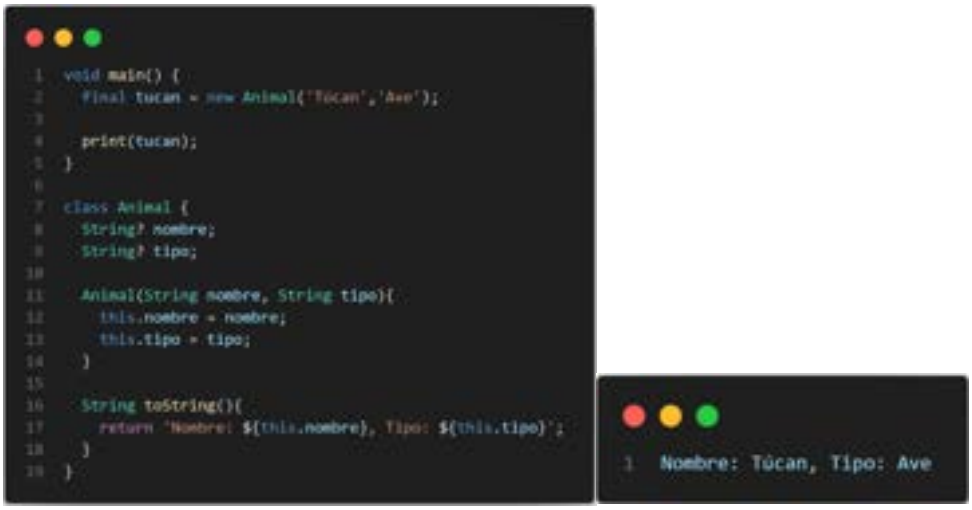


```dart
void main() {
  final tucan = new Animal('Tucan','Ave');

  print(tucan.nombre);
  print(tucan.tipo);
}

class Animal {
  String? nombre;
  String? tipo;

  Animal(String nombre, String tipo){
    this.nombre = nombre;
    this.tipo = tipo;
  }
}
```

```
Tucán
Ave
```

Utilizando lo aprendido sobre funciones en Dart, se puede sobrescribir el método **toString**() para retornar los atributos de la clase. Este método se ejecuta por defecto cuando se imprime el objeto o se invoca directamente.

Nota: Para evitar errores al escribir el nombre de un atributo compuesto junto con la palabra reservada **this**, es necesario encapsularlo entre llaves { }.



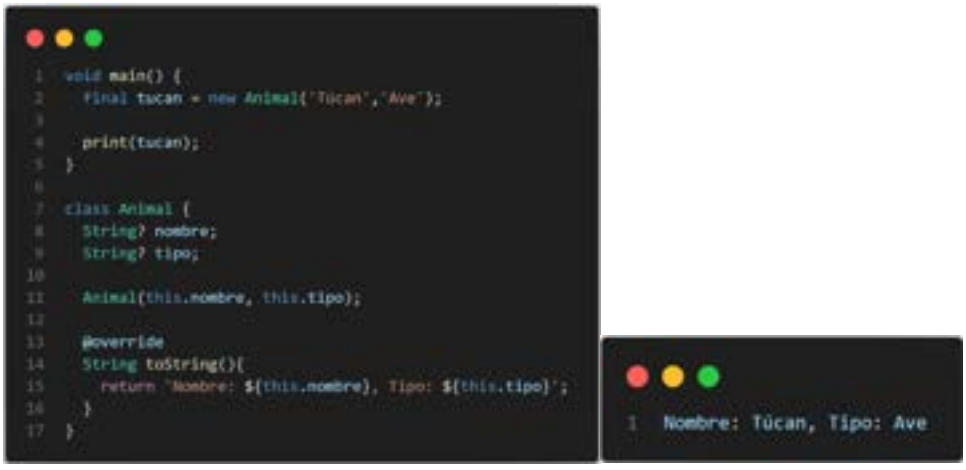
```dart
void main() {
  final tucan = new Animal('Tucan','Ave');

  print(tucan);
}

class Animal {
  String? nombre;
  String? tipo;

  Animal(String nombre, String tipo){
    this.nombre = nombre;
    this.tipo = tipo;
  }

  String toString(){
    return 'Nombre: ${this.nombre}, Tipo: ${this.tipo}';
  }
}
```

```
Nombre: Tucan, Tipo: Ave
```

Nota: Al revisar el código, se pueden notar algunas palabras resaltadas que, aunque no representan errores, sugieren mejoras según las recomendaciones del lenguaje. En este caso, es posible inicializar los atributos de la clase de una manera más formal cuando sea viable, como se muestra a continuación.

Algunas mejoras implementadas respecto a lo visto anteriormente son:

- Eliminar la palabra reservada new al crear un objeto, ya que Dart infiere automáticamente la instanciación.
- Utilizar una inicialización formal en el constructor de la clase, asignando directamente los parámetros a los atributos con this, siempre que los nombres de los parámetros coincidan con los atributos. Esto proporciona una sintaxis más clara y concisa.
- Indicar cuando los métodos de las superclases son sobrescritos, utilizando la anotación @override.
- Eliminar el uso de this en el método toString cuando no es estrictamente necesario dentro del contexto.



```dart
void main() {
  final tucan = new Animal('Tucan','Ave');

  print(tucan);
}

class Animal {
  String? nombre;
  String? tipo;

  Animal(this.nombre, this.tipo);

  @override
  String toString(){
    return 'Nombre: ${this.nombre}, Tipo: ${this.tipo}';
  }
}
```

```
Nombre: Tucan, Tipo: Ave
```

De manera similar, se pueden utilizar parámetros por nombre en el constructor. Para ello, es necesario empaquetar los parámetros entre llaves { } y luego asignarlos a los atributos de la clase. Esto permite mayor flexibilidad al momento de instanciar el objeto en el main, dado que los argumentos pueden ser proporcionados en cualquier orden, siempre que se especifique el nombre del parámetro correspondiente.

```
void main() {
  final tucan = new Animal(nombre: 'Túcan',tipo: 'Ave');

  print(tucan);
}

class Animal {
  String? nombre;
  String? tipo;

  Animal({this.nombre, this.tipo});

  @override
  String toString(){
    return 'Nombre: ${this.nombre}, Tipo: ${this.tipo}';
  }
}
```

```
Nombre: Túcan, Tipo: Ave
```

Nota: Se eliminó el operador ? utilizado para controlar valores null en los atributos de la clase. Como resultado, se generó un error en la inicialización de estos dentro del constructor, ya que Dart requiere que todas las variables no anulables sean inicializadas antes de su uso.

```
void main() {
  final tucan = new Animal(nombre: 'Túcan',tipo: 'Ave');

  print(tucan);
}

class Animal {
  String nombre;
  String tipo;

  Animal({this.nombre, this.tipo});

  @override
  String toString(){
    return 'Nombre: ${this.nombre}, Tipo: ${this.tipo}';
  }
}
```

```
Nombre: Túcan, Tipo: Ave
```

Para solucionar el error y garantizar que los atributos sean inicializados correctamente, se puede utilizar la palabra reservada **required**. Esto obliga a que los valores sean proporcionados al momento de instanciar la clase, asegurando así su correcta inicialización y evitando problemas relacionados con **null**.

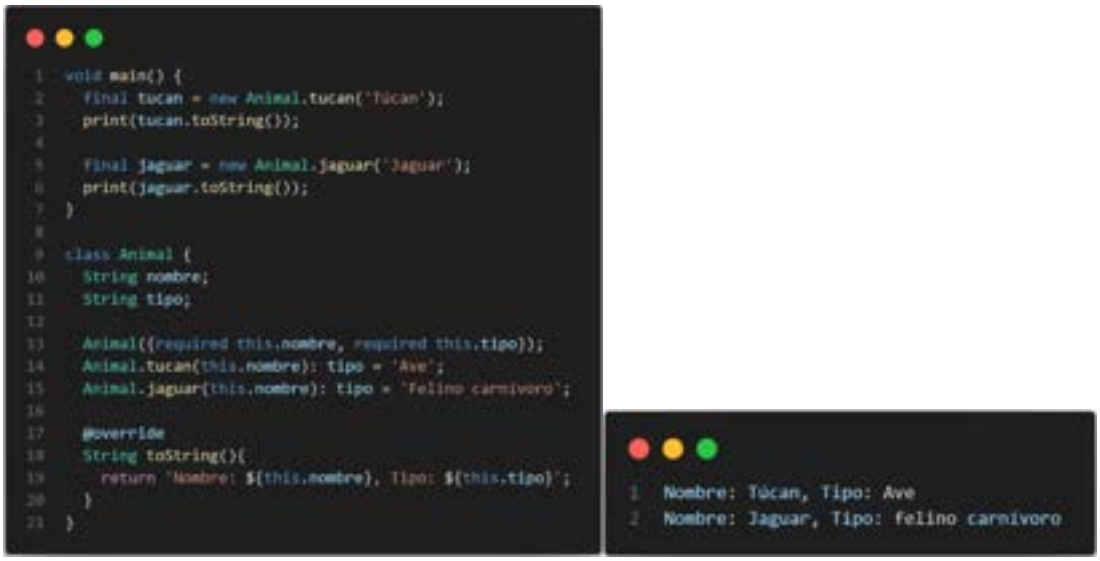
```dart
void main() {
  final tucan = new Animal(nombre: 'Tucan',tipo: 'Ave');

  print(tucan);
}

class Animal {
  String nombre;
  String tipo;

  Animal({required this.nombre, required this.tipo});

  @override
  String toString(){
    return 'Nombre: ${this.nombre}, Tipo: ${this.tipo}';
  }
}
```

```
Nombre: Tucan, Tipo: Ave
```

5.2.3 Constructores nombrados

Los constructores con nombre permiten definir inicializaciones más específicas dentro de una clase. En Dart, se pueden utilizar los dos puntos : después de los paréntesis del constructor para asignar valores a las propiedades de la clase antes de que se ejecute el cuerpo del constructor.

Esto es especialmente útil para el control de **Sound Null Safety**, ya que garantiza que los atributos sean inicializados en el momento adecuado. Si no se hace de esta manera, Dart intentaría instanciar primero el objeto sin inicializar las propiedades, lo que generaría un error debido a la falta de valores requeridos.



```dart
void main() {
  final tucan = new Animal.tucan('Tucan');
  print(tucan.toString());

  final jaguar = new Animal.jaguar('Jaguar');
  print(jaguar.toString());
}

class Animal {
  String nombre;
  String tipo;

  Animal({required this.nombre, required this.tipo});
  Animal.tucan(this.nombre): tipo = 'Ave';
  Animal.jaguar(this.nombre): tipo = 'Felino carnivoro';

  @override
  String toString(){
    return 'Nombre: ${this.nombre}, Tipo: ${this.tipo}';
  }
}
```

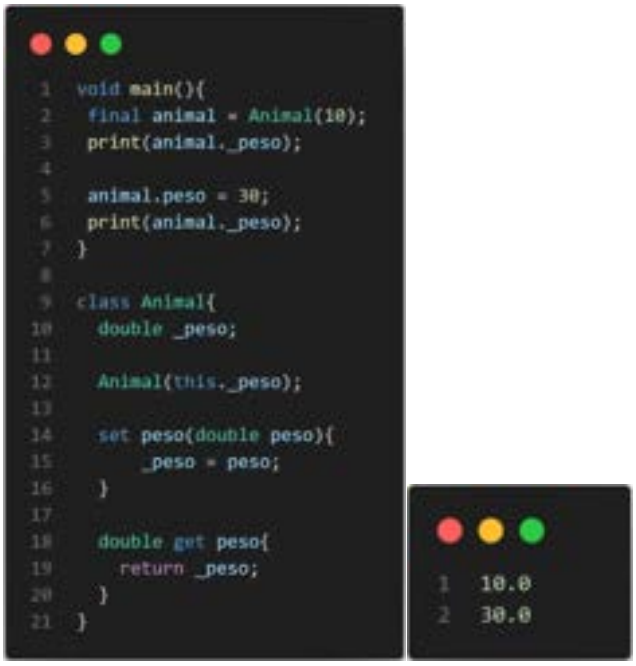
```
Nombre: Tucan, Tipo: Ave
Nombre: Jaguar, Tipo: felino carnivoro
```

5.3 Getters and Setters

Los métodos **getter** y **setter** en Dart permiten acceder y modificar los atributos de una clase de manera controlada. Los **getters** se utilizan para obtener el valor de una propiedad, mientras que los **setters** permiten asignar un nuevo valor a dicha propiedad.

Estos métodos proporcionan un mecanismo seguro para manipular los atributos de una clase, asegurando la integridad de los datos y permitiendo aplicar validaciones o restricciones cuando sea necesario.

Por **buenas prácticas**, los getters y setters deben seguir la sintaxis adecuada en Dart, utilizando las palabras clave **get** y **set**, respectivamente.



```dart
void main(){
  final animal = Animal(10);
  print(animal._peso);

  animal.peso = 30;
  print(animal._peso);
}

class Animal{
  double _peso;

  Animal(this._peso);

  set peso(double peso){
      _peso = peso;
  }

  double get peso{
    return _peso;
  }
}
```

```
10.0
30.0
```

En Dart, se puede establecer la **visibilidad** de las propiedades utilizando un **guion bajo (_)** antepuesto al nombre del atributo. Esto indica que la propiedad es **privada** y solo puede ser accedida dentro de la misma **clase** o **archivo** en el que se define.

Esto significa que los atributos con guion bajo no pueden ser accedidos directamente desde fuera de la clase, lo que **refuerza el encapsulamiento** y promueve el uso de **getters y setters** para interactuar con ellos de manera controlada.

En resumen, **una propiedad privada solo es accesible dentro de su clase**, y si se requiere acceder a ella desde fuera, se deben definir métodos **getter** y **setter** que permitan su consulta o modificación de manera segura.

```dart
double _peso;
```

Dart es una plataforma que utiliza setters, métodos especiales para identificar y asignar nuevos valores a propiedades privadas de una clase, adecuados por un parámetro que permita validaciones y restricciones antes de modificar el valor de un atributo.

```dart
set peso(double peso){
    _peso = peso;
}
```

Dart utiliza getters para identificar propiedades sin paréntesis () y permite un acceso seguro a sus valores. Facilitan la encapsulación y evitan modificaciones directas. Los getters también permiten calcular dinámicamente valores basados en otros atributos de clase, lo que garantiza la seguridad.

```
18    double get peso{
19      return _peso;
20    }
```

Por último, es importante notar que los métodos especiales *getters* y *setters* tienen una peculiaridad: se accede a ellos de la misma forma que a un parámetro, sin necesidad de utilizar paréntesis.

```
5     animal.peso = 30;
6     print(animal._peso);
```

Nota: Dentro de los métodos *getters* y *setters* se puede incorporar lógica adicional para validar y controlar los valores asignados. Por ejemplo, en un *setter* se puede asegurar que un número sea mayor que cero antes de ser almacenado, evitando así valores no deseados o incorrectos en la propiedad.

```
void main(){
  final animal = Animal(30);
  print(animal._peso);

  animal.peso = 0;
  print(animal._peso);
}

class Animal{
  double _peso;

  Animal(this._peso);

  set peso(double peso){
    if(peso <= 0){
      throw Exception('El peso no puede ser menor o igual a 0');
    }
    _peso = peso;
  }

  double get peso{
    return _peso;
  }
}
```

```
1   10.0
```

```
Exception has occurred.
_Exception (Exception: El peso no puede ser menor o igual a 0)
```

Si es posible y se desea, se puede utilizar la función flecha (=>) para simplificar la sintaxis del código, especialmente en *getters*, ya que permiten retornar valores en una sola línea, haciendo el código más compacto y legible.

```
21    double get peso => _peso;
```

5.3.1 Aserciones (aseert)

En Dart, las *aserciones* son declaraciones utilizadas para verificar que ciertas condiciones se cumplan durante la ejecución del programa. Su principal función es ayudar a detectar y diagnosticar errores o comportamientos inesperados, facilitando la depuración del código.

La estructura de una aserción en Dart es: **assert**(condición, mensaje);. La **condición** debe ser una expresión booleana, y en caso de que no se cumpla, se mostrará el **mensaje** especificado. Este tipo de validación también se puede utilizar dentro de funciones para garantizar que se cumplan ciertos criterios durante la ejecución del programa.

```dart
void main() {
  final animal = Animal(10);
  print(animal._peso);

  animal.peso = -1;
  print(animal._peso);
}

class Animal {
  double _peso;

  Animal(this._peso);

  set peso(double nuevoPeso) {
      assert(nuevoPeso > 0, 'El peso debe ser mayor a 0');
      _peso = peso;
  }

  double get peso => _peso;
}
```

```
1  10.0
```

```
Failed assertion: line 15 pos 14: 'nuevoPeso > 0': E
peso debe ser mayor a 0)
```

También es posible utilizar las aserciones (assert) dentro de los constructores para realizar validaciones al momento de crear un objeto. Esto permite garantizar que los valores iniciales cumplan ciertas condiciones antes de que la instancia sea creada correctamente.

```dart
void main(){
  final ciudad = Ciudad(nombre: '',
  poblacion: -1);

  print(ciudad.nombre);
}

class Ciudad{
  final String nombre;
  final int poblacion;

  Ciudad({
    required this.nombre,
    required this.poblacion
  }):assert(nombre.isNotEmpty, 'El nombre no puede estar vacio'),
    assert(poblacion >= 0, 'La poblacion no puede ser negativa');
}
```

Al ejecutar el código anterior, se lanzará una excepción, teniendo en cuenta que el valor asignado al nombre de la ciudad es una cadena vacía, lo cual no cumple con la validación definida.

```
15    }):assert(nombre.isNotEmpty, 'El nombre no puede estar vacio'),

Exception has occurred.
_AssertionError ('file:///C:/Users/USUARIO/OneDrive/DOC
Failed assertion: line 15 pos 13: 'nombre.isNotEmpty': El nombre no puede
estar vacio)
```

Para evitar la excepción y garantizar un funcionamiento correcto, es necesario ajustar los argumentos proporcionados al momento de instanciar el objeto, asegurando que cumplan con las validaciones establecidas.

```dart
void main(){
  final ciudad = Ciudad(nombre: 'Florencia',
  poblacion: 117946);

  print('La ciudad: ${ciudad.nombre} tiene ${ciudad.poblacion}');
}
```

5.4 Herencia

La herencia es un pilar fundamental de la programación orientada a objetos, ya que permite crear nuevas clases a partir de otras ya existentes, reutilizando y ampliando su funcionalidad. En Dart, este mecanismo se implementa utilizando la palabra clave `extends` en la declaración de la clase. De esta forma, la nueva clase —conocida como subclase— hereda los atributos y métodos de la clase base o clase padre. Además, puede añadir nuevos métodos o sobrescribir los heredados según sea necesario.

Es importante tener en cuenta que Dart no permite herencia múltiple; es decir, una clase solo puede extender de una única clase padre. Para acceder a los miembros de la clase padre desde la subclase, se utiliza la palabra clave `super`.

```dart
class Animal{
  String color;

  Animal(this.color);

  void hacerSonido(){
    print("Hace sonido");
  }
}

class jaguar extends Animal{
  jaguar(String color):super(color);

  @override
  void hacerSonido(){
    print("El jaguar está rugiendo");
  }
}

class BoaConstrictor extends Animal{
  BoaConstrictor(String color):super(color);

  @override
  void hacerSonido(){
    print("La Boa está siseando");
  }
}
```

Se puede observar que, aunque tanto el jaguar como la boa tienen un método con el mismo nombre, este ha sido sobrescrito en cada clase, lo que permite que su comportamiento sea distinto en cada caso.

```dart
void main(){
  Jaguar jaguar = Jaguar("Amarillo y manchas negras");
  jaguar.hacerSonido();

  BoaConstrictor boa = BoaConstrictor("Café bronceado");
  boa.hacerSonido();
}
```

```
El jaguar está rugiendo
La Boa está siseando
```

5.4.1 Clase abstracta

En Dart, una clase abstracta es aquella que no puede ser instanciada directamente, lo que significa que no se pueden crear objetos a partir de ella. Su propósito principal es servir como una plantilla para otras clases, permitiendo definir una estructura común que será compartida y personalizada por las clases que la extiendan.

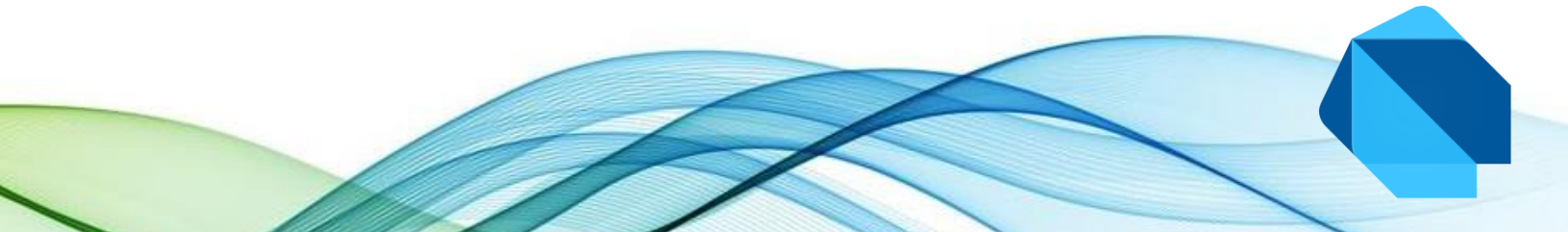
Estas clases pueden incluir métodos abstractos, es decir, métodos sin implementación que obligan a las subclases a proporcionar su propio comportamiento. Esto favorece la reutilización de código y la coherencia entre diferentes clases relacionadas.

Para comprobar esto en la práctica, se puede crear una clase llamada **Animal**, marcándola como abstracta con la palabra clave **abstract**. Luego, al intentar instanciar un objeto directamente desde **Animal**, Dart generará un error, evidenciando que no se permite la creación de instancias a partir de clases abstractas.

```dart
void main() {
  final tucan = new Animal();
}

abstract class Animal {
  String? color;

  void sonido();
}
```

```
is abstract and can't be instantiated.
  final tucan = new Animal();
                ^^^^^^
```

Al ejecutar el código, se genera un error que indica que no es posible instanciar la clase. Esto ocurre porque, como ya se ha mencionado, una de las características fundamentales de las clases abstractas en Dart es que no pueden ser instanciadas directamente.



Dado que una clase abstracta funciona como una plantilla, se puede crear otra clase que la implemente. Para ello, basta con utilizar la palabra clave **implements** seguida del nombre de la clase abstracta al momento de declarar la nueva clase. De esta forma, la clase hija asume el compromiso de definir todos los métodos y propiedades establecidos en la clase abstracta.

```dart
void main() {
  final tucan = new Animal();
}

abstract class Animal {
  String? color;

  void sonido();
}

class Tucan  implements Animal{}
```

Sin embargo, aún se pueden observar errores en la clase **Tucan**. Esto se debe a que Dart indica que no se han implementado los métodos definidos en la clase abstracta Animal. Dado que la palabra clave **implements** obliga a proporcionar una implementación concreta de todos los métodos y propiedades de la clase abstracta, es necesario definirlos dentro de **Tucan** para corregir el error.

```
AbstractV1.dart:8:8: Context: 'Animal.sonido' is defined here.
  void sonido();
       ^^^^^^
```

Esto se debe a que la clase **Tucan** debe definir un color y contar con un sonido, además de implementar cualquier otro elemento requerido por la plantilla abstracta.

Aplicando buenas prácticas, como el uso de la anotación **@override** en los métodos sobrescritos, el control de valores no nulos y la implementación completa de los elementos definidos en la clase abstracta, se obtiene un código más estructurado.

En este nuevo enfoque, en lugar de instanciar la clase abstracta, se crea un objeto a partir de la clase concreta derivada de ella. De esta manera, es posible generar múltiples clases a partir de la plantilla abstracta, garantizando coherencia y reutilización del código.

```dart
void main() {
  final tucan = new Tucan('Amarillo y negro');
  print(tucan.color);
  tucan.sonido();
}

abstract class Animal {
  String? color;

  Animal(this.color);

  void sonido();
}

class Tucan implements Animal {
  @override
  String? color;

  Tucan(this.color);

  @override
  void sonido() => print('Graznido');
}
```

```
Amarillo y negro
Graznido
```

5.4.2 Mixins

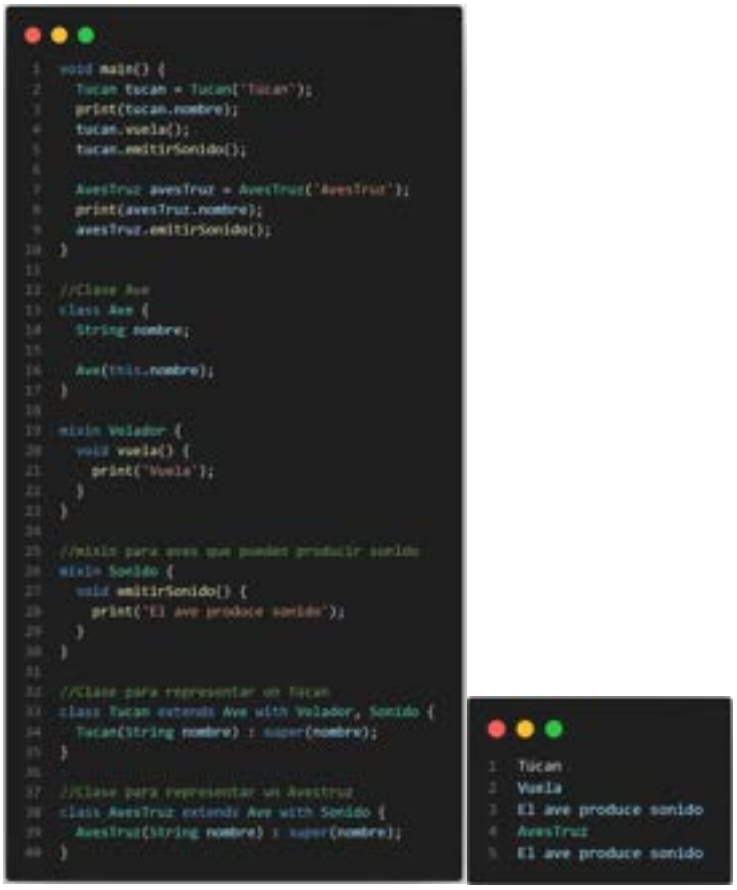
Los mixins en Dart permiten reutilizar código en múltiples jerarquías de clases sin necesidad de recurrir a la herencia múltiple, la cual no está permitida en este lenguaje para evitar ambigüedades y conflictos de nombres. Para aplicar un mixin, se utiliza la palabra clave **with**, seguida del o los mixins que se desean incluir (*Mixins*, s.f.).

Esta característica representa una alternativa poderosa a la herencia tradicional, ya que permite que una clase adquiera comportamientos de múltiples fuentes sin extender directamente de todas ellas. Es importante tener en cuenta que los mixins no pueden ser instanciados y, por tanto, no pueden tener constructores.

A modo de ejemplo, se define una clase base llamada Ave, que contiene el atributo nombre. Luego, se crean dos mixins: Volador y Sonido, cada uno con un método específico que imprime un mensaje relacionado con su funcionalidad.

Posteriormente, se implementan dos clases: Tucan y Avestruz. Ambas extienden de la clase Ave, pero, además, el Tucan aplica los dos mixins (Volador y Sonido), mientras que el Avestruz solo utiliza el mixin Sonido.

Finalmente, se crean los respectivos objetos y se invocan los métodos disponibles para cada uno, demostrando cómo se combinan la herencia y los mixins para construir clases más flexibles y reutilizables.



```
Tucan
Vuela
El ave produce sonido
AvesTruz
El ave produce sonido
```

En el ejemplo anterior, se define una clase Ave, encargada de almacenar el nombre de un ave. Sin embargo, a partir de esta se generan clases más especializadas, como Tucan y Avestruz, que combinan herencia y *mixins* para extender su funcionalidad.

Por un lado, la clase Tucan hereda de Ave, pero además incorpora los *mixins* Volador y Sonido, lo que le permite acceder a los métodos definidos en ellos. De manera similar, la clase Avestruz también hereda de Ave, pero en este caso solo hace uso del *mixin* Sonido, lo que le otorga únicamente la funcionalidad relacionada con la emisión de sonidos.

En la función main, se crean los respectivos objetos y se ejecutan los métodos disponibles para cada clase, demostrando cómo los *mixins* permiten agregar comportamientos adicionales sin necesidad de modificar la estructura de la jerarquía de clases.

Además, los *mixins* pueden combinarse con la herencia tradicional y aplicarse sobre clases concretas o abstractas, brindando mayor flexibilidad en el diseño del código.

```dart
// Mixin para el enfoque filosófico de la ética
mixin EticaFilosofica {
  void discutirEtica() {
    print('Discutiendo cuestiones éticas y morales.');
  }
}

// Mixin para el enfoque filosófico de la lógica
mixin LogicaFilosofica {
  void usarLogica() {
    print('Utilizando la lógica y el razonamiento.');
  }
}

// Mixin para el enfoque filosófico de la epistemología
mixin EpistemologiaFilosofica {
  void reflexionarEpistemologia() {
    print('Reflexionando sobre la naturaleza del conocimiento.');
  }
}

// Clase abstracta para representar a un filósofo antiguo
abstract class FilosofoAntiguo {
  String nombre;

  FilosofoAntiguo(this.nombre);

  void reflexionar() {
    print('$nombre está reflexionando:');
  }
}

// Clase que representa a un filósofo con enfoque filosófico
class Filosofo extends FilosofoAntiguo
    with EticaFilosofica, LogicaFilosofica, EpistemologiaFilosofica {
  Filosofo(String nombre) : super(nombre);
}

void main() {
  var socrates = Filosofo('Sócrates');
  socrates.reflexionar();
  socrates.discutirEtica();
  socrates.usarLogica();
  socrates.reflexionarEpistemologia();
}
```

```
Sócrates está reflexionando:
Discutiendo cuestiones éticas y morales.
Utilizando la lógica y el razonamiento.
Reflexionando sobre la naturaleza del conocimiento.
```

5.4.3 Métodos de extensión

Los métodos de extensión son una característica avanzada de Dart que permiten agregar métodos y propiedades a tipos existentes sin modificar su código original. Esta funcionalidad resulta especialmente útil cuando se necesita ampliar el comportamiento de una clase sin acceso a su implementación o sin alterar su estructura (*Extension Types*, s.f.).

Para definir una extensión, se utiliza la palabra clave **extension**, seguida de un nombre opcional. Además, la palabra clave **on** indica el tipo de dato al que se aplicará la extensión. También es posible definir niveles de acceso en estas extensiones.

En el siguiente ejemplo, se aplica esta técnica sobre el tipo String, creando una extensión llamada ExtensionString. Esta extensión añade dos métodos nuevos: uno para contar la cantidad de vocales en una cadena de texto y otro que imprime el contenido en mayúsculas.

```dart
// Definimos una extensión llamada StringExtension
extension on String {
  // Método de extensión para contar las vocales en una cadena
  int contarVocales() {
    final vocales = ['a', 'e', 'i', 'o', 'u', 'á', 'é', 'í', 'ó', 'ú'];
    return this.split('').where((char) => vocales.contains(char)).length;
  }

  void imprimeMayuscula() => print(toUpperCase());
}

void main(List<String> args) {
  String texto = 'Hola, ¿cómo te encuentras hoy?';

  // Utilizamos la extensión para contar las vocales en el texto
  int cantidadVocales = texto.contarVocales();

  print('El texto contiene $cantidadVocales vocales.');
  texto.imprimeMayuscula();
}
```

```
El texto contiene 10 vocales.
HOLA, ¿CÓMO TE ENCUENTRAS HOY?
```

Como se puede observar, la funcionalidad del tipo String se ha ampliado mediante dos métodos adicionales sin modificar su código original. Estos métodos se integran de manera natural con el comportamiento estándar de String, permitiendo extender sus capacidades sin afectar su implementación base.

6 FUTURE

Un Future representa el resultado o la promesa de una operación asíncrona que se completará en un momento futuro, proporcionando un valor que puede ser utilizado posteriormente. Sin embargo, es fundamental considerar que la ejecución puede fallar, por lo que el manejo adecuado de excepciones es esencial (*Asynchronous Programming: Futures, Async, Await*, s.f.).

A continuación, se presenta un ejemplo que ilustra el proceso de inscripción en el sistema de una universidad. En este caso, se realiza una solicitud asíncrona para registrar materias utilizando el método Future.delayed. Este método introduce un tiempo de espera antes de ejecutar la función, simulando así un posible retraso en la operación.

```dart
void main() {
  // Mensaje inicial antes de comenzar la operación asíncrona.
  print('Inicio de proceso de matrícula en el sistema de la Universidad');

/* Llamada al método `httpGet` que simula una solicitud HTTP al sistema
   de matrículas y espera su respuesta.*/
  httpGet("http://sistema/matriculas").then((value) {
    print(value);
  });

  // Mensaje indicando que la operación está en proceso.
  print("¡Procesando matrícula!");
}

// Este método toma un [url] como parámetro y retorna un `Future<String>`
Future<String> httpGet(String url) {
  // Simula un retraso de 3 segundos antes de completar el `Future`.
  return Future.delayed(const Duration(seconds: 3), () {
    return "¡Asignaturas matriculadas con éxito!";
  });
}
```

En este ejemplo, se observa cómo el proceso de matrícula inicia con un mensaje informativo, seguido de otro que indica que la matrícula está en proceso. Tres segundos después, el Future devuelve el valor prometido, confirmando que las asignaturas han sido matriculadas exitosamente.

Es importante destacar que, si el código se ejecutara de forma síncrona, el flujo sería diferente: primero se mostraría el inicio del proceso, luego el mensaje de matrícula exitosa y, finalmente, la indicación de que la matrícula está en proceso. Esto evidencia cómo la asincronía modifica la secuencia de ejecución, permitiendo que el programa continúe su funcionamiento mientras espera la finalización de una tarea.

```
Inicio de proceso de matrícula en el sistema de la Universidad
¡Procesando matrícula!
¡Asignaturas matriculadas con éxito!
```

6.1 Manejo excepciones Future

Partiendo del ejercicio anterior, se introduce una excepción dentro del método Future mediante el uso de **throw**, mientras que la línea de **return** se comenta para evitar una alerta de sintaxis causada por código inalcanzable.

Además, al invocar el método **httpGet**, se utiliza **catchError** para capturar y manejar la excepción de manera controlada. Esto permite ilustrar cómo gestionar errores en operaciones asíncronas mediante un manejo explícito de excepciones. En este contexto, una excepción representa un evento inesperado que interrumpe el flujo normal del programa durante su ejecución.

```dart
void main() {
  // Mensaje inicial antes de comenzar la operación asíncrona.
  print('Inicio de proceso de matrícula en el sistema de la Universidad');

/* Llamada al método `httpGet` que simula una solicitud HTTP al sistema
   de matrículas y espera su respuesta.*/
  httpGet("http://sistema/matriculas").then((value) {
    print(value);
  }).catchError((err) {
    print("Se presento un error: $err");
  });
  // Mensaje indicando que la operación está en proceso.
  print("¡Procesando matricula!");
}

// Este método toma un [url] como parámetro y retorna un `Future<String>`
Future<String> httpGet(String url) {
  // Simula un retraso de 3 segundos antes de completar el `Future`.
  return Future.delayed(const Duration(seconds: 3), () {
    //forzamos una excepción con la palabra clave throw
    throw "Error en el proceso de matricula!";
    //return "¡Asignaturas matriculadas con exito!";
  });
}
```

En esta salida, se observa el mensaje generado por la excepción forzada con **throw**, en lugar del valor de retorno que se habría obtenido si la promesa del Future se hubiera completado con éxito. Este ejemplo demuestra cómo utilizar de manera combinada los métodos **then** y **catchError**, permitiendo gestionar tanto los resultados exitosos como los posibles errores que puedan surgir durante la ejecución del Future.

```
Inicio de proceso de matrícula en el sistema de la Universidad
¡Procesando matricula!
Se presento un error: Error en el proceso de matricula!
```

6.2 Async – Await

Las palabras clave **async** y **await** son fundamentales para trabajar con Future en Dart. Al declarar una función con **async**, se indica que es asíncrona y devuelve un Future, incluso si no se especifica explícitamente en el código. Por otro lado, **await** se usa dentro de funciones asíncronas para pausar la ejecución hasta obtener el resultado de una operación sin bloquear el flujo general del programa.

Al reorganizar el código para simular una solicitud en un sistema de matrículas utilizando **async** y **await**, el resultado se vuelve más intuitivo y fácil de comprender. En este enfoque, el método **httpGet** se declara como asíncrono mediante la palabra clave **async**, lo que permite utilizar **await** para esperar su resultado de manera más clara y estructurada.
Además, la función **main** también se define como asíncrona, lo que facilita el uso de **await** dentro de ella y mejora la integración de esta técnica en la lógica del programa.

```dart
void main() async {
  print("Inicio del programa de matriculas...");
  // Espera el resultado del Future
  String datos = await httpGet();
  print(datos);

  print("Fin del programa de matriculas");
}

Future<String> httpGet() async {
  print("Realizando solicitud al servidor de matriculas...");
  // Simula un retraso de 3 segundos
  await Future.delayed(Duration(seconds: 3));
  // Devuelve el resultado de forma asincrona
  return "Datos obtenidos del servidor!!!";
}
```

Durante la ejecución del programa, primero se muestra el mensaje **"Inicio del programa de matrículas"**, seguido inmediatamente por **"Realizando solicitud al servidor de matrículas"**. Luego, se introduce un retraso de 3 segundos, tras el cual aparece el mensaje **"Datos obtenidos del servidor"**. Finalmente, el programa concluye con la impresión de **"Fin del programa de matrículas"**.

```
Inicio del programa de matriculas...
Realizando solicitud al servidor de matriculas...
Datos obtenidos del servidor!!!
Fin del programa de matriculas
```

6.3 Try – catch – finally – on

Las estructuras **try**, **catch**, **finally** y **on** son herramientas fundamentales en los lenguajes de programación para el manejo de excepciones. Su propósito es controlar posibles errores, ya sean esperados o inesperados, que puedan surgir durante la ejecución del código (*Error Handling*, s.f.).

- **try**: Se utiliza para encerrar el código que podría generar una excepción.
- **catch**: Se ejecuta si ocurre una excepción no identificada o no controlada.
- **finally**: Se ejecuta siempre, independientemente de si la ejecución del bloque **try** fue exitosa o no.

```dart
void main() async {
  print("Inicio del programa de matriculas...");
  try {
    /* El bloque de código que podria generar una excepción, pero
       que intenta ejecutar porque se espera que se ejecute sin problema*/
    // Espera el resultado del Future
    String datos = await httpGet();
    print(datos);
  } catch (e) {
    // Se ejecuta cuando ocurre una excepción genérica
    print("Se presento un error: $e");
  } finally {
    // Se ejecuta independientemente si funciono o no el try
    print("Cierre de solicitud al servidor");
    print("Fin del programa de matriculas");
  }
}

Future<String> httpGet() async {
  print("Realizando solicitud al servidor de matriculas...");
  // Simula un retraso de 3 segundos
  await Future.delayed(Duration(seconds: 3));
  // Devuelve el resultado de forma asincrona
  throw Exception("No se encontro la URL");
  //return "Datos obtenidos del servidor!!!";
}
```

En el ejemplo anterior, cuando se produce una excepción dentro del bloque **try**, esta es capturada y gestionada por el **catch**, que muestra un mensaje indicando el error. Posteriormente, el programa continúa su ejecución y se ejecuta el bloque **finally**, garantizando que se realicen las acciones finales sin importar si hubo o no una excepción.

```
Inicio del programa de matriculas...
Realizando solicitud al servidor de matriculas...
Se presento un error: Exception: No se encontro la URL
Cierre de solicitud al servidor
Fin del programa de matriculas
```

Por último, la estructura **on** permite manejar excepciones específicas cuando se conoce su tipo y origen, como se muestra a continuación. Es importante destacar que, donde se usa la palabra **Exception** después de **on**, esta puede ser reemplazada por el nombre exacto de la excepción que se desea capturar, permitiendo un control más preciso de los errores en el programa.

```dart
void main() async {
  print("Inicio del programa de matriculas...");
  try {
    /* El bloque de código que podría generar una excepción, pero
      que intenta ejecutar porque se espera que se ejecute sin problema*/
    // Espera el resultado del Future
    String datos = await httpGet();
    print(datos);
  } on Exception catch (e) {
    //El on controla las throw Exception
    print("Se presento la $e");
  } catch (e) {
    // controla los throw
    // Se ejecuta cuando ocurre una excepción genérica
    print("Se presento un error: $e");
  } finally {
    // Se ejecuta independientemente si funciono o no el try
    print("Cierre de solicitud al servidor");
    print("Fin del programa de matriculas");
  }
}

Future<String> httpGet() async {
  print("Realizando solicitud al servidor de matriculas...");
  // Simula un retraso de 3 segundos
  await Future.delayed(Duration(seconds: 3));
  // Devuelve el resultado de forma asincrona
  throw Exception("No se encontro la URL");
  //return "Datos obtenidos del servidor!!!";
}
```

Al ejecutar el ejercicio anterior, se puede observar que la instrucción **throw Exception** genera una excepción que es correctamente capturada por la estructura **on**. Esto demuestra cómo se pueden manejar errores específicos de manera controlada dentro del flujo del programa.

```
Inicio del programa de matriculas...
Realizando solicitud al servidor de matriculas...
Se presento la Exception: No se encontro la URL
Cierre de solicitud al servidor
Fin del programa de matriculas
```

Si en lugar de usar **throw Exception** solo se deja **throw**, la excepción será capturada por el bloque **catch** en lugar del **on**. Esto ocurre porque **throw** sin especificar un tipo de excepción genera un error de tipo **dynamic**, el cual no coincide con la condición establecida en **on** (que espera un tipo específico de excepción). En este caso, el **catch** actuará como una captura general de cualquier excepción no manejada explícitamente.

```
Inicio del programa de matriculas...
Realizando solicitud al servidor de matriculas...
Se presento un error: No se encontro la URL
Cierre de solicitud al servidor
Fin del programa de matriculas
```

7 AGRADECIMIENTO

Manifestamos nuestro más sincero agradecimiento a la Universidad de la Amazonía por la formación de los fundamentos para obtener los conocimientos que facilitaron la creación de esta obra. Valoramos al programa de Ingeniería de Sistemas por su respaldo constante y por promover un entorno de aprendizaje e innovación constante.

Nuestra gratitud también se extiende a los colegas y estudiantes, quienes han sido una fuente constante de inspiración y motivación para buscar maneras de compartir el conocimiento. Sus preguntas, comentarios y discusiones han enriquecido este trabajo, ayudándonos a profundizar nuestro conocimiento de Dart y a presentar conceptos de manera clara y accesible.

Por último, agradecemos a nuestra familia y amigos por su respaldo durante el proceso de redacción de este libro.

8 BIBLIOGRAFÍA

Asynchronous programming: futures, async, await. (s.f.). Dart. Recuperado Marzo 6, 2025, de https://dart.dev/libraries/async/async-await

Classes. (s.f.). Dart. Recuperado Abril 25, 2024, de https://dart.dev/language/classes

Dart2native. (s.f.). Dart. Recuperado Enero 03, 2024, de https://dart.cn/tools/dart2native

Dart doc. (s.f.). Dart. Recuperado Marzo 2, 2024, de https://dart.dev/tools/dart-doc

Dart overview. (s.f. -a). Dart. Recuperado Marzo 2, 2025, de https://dart.dev/overview

Effective DART: documentation. (s.f.). Dart. Recuperado Febrero 10, 2024, de https://dart.dev/effective-dart/documentation

Dart overview. (s.f.-b). Dart. Recibido Enero 1, 2024, de https://dart.dev/overview#nativeplatform

Effective Dart: style. (s.f.). Dart. Recuperado Enero 4, 2025, de https://dart.dev/effective-dart/style

Error handling. (s.f.). Dart. Recibido Marzo 7, 2025, de https://dart.dev/language/error-handling

Extension types. (s.f.). Dart. Recuperado Febrero 12, 2025, de https://dart.dev/language/extension-types

file_names. (s.f.). Dart. Recuperado Enero 4, 2024, de https://dart.dev/tools/linter-rules/file_names

Functions. (s.f.). Dart. Recuperado Marzo 20, 2024, de https://dart.dev/language/functions

Google for Developers. (2012, Junio 28). *Google I/O 2012 - DART - A Modern Web Language* [Video]. YouTube. https://www.youtube.com/watch?v=bsGgfUreyZw

Mixins. (s.f.). Dart. Recuperado Abril 26, 2025, de https://dart.dev/language/mixins

Tools. (s.f.). Dart. Recuperado Enero 01, 2024, de https://dart.dev/tools

Variables. (s.f.). Dart. Recuperado Marzo 1, 2024, de https://dart.dev/language/variables

EDITORIAL CIIDES

978-628-95694-8-3