

Universidad Nacional Autónoma de México.  
Facultad de Ingeniería.

Sistemas Operativos.

**Documento previo a exposición.**

***Concurrencia y programación asíncrona en Dart.***

Profesor. Gunnar Wolf.

Niver Martínez.  
Jorge Barcenás.

Cd. Universitaria, a 13 de marzo de 2020.

” ¿Quién puede esperar tranquilamente mientras el barro se asienta?

¿Quién puede permanecer en calma hasta el momento de actuar?”

- Laozi, Tao Te Ching (Haverbeke, M., 2018)

## **Resumen.**

La programación asíncrona y la concurrencia son dos conceptos que suele ser planteados por cuando nos involucramos con situaciones que pueden ejecutar los procesos de una manera distinta a como nos acomoda la programación, ya que muchas veces pensamos de manera muy secuencial y estructurada, lo cual no tiene por qué ser ni malo ni bueno. Sin embargo, el manejo de eventos en programación requiere pensar dadas algunas condiciones casi de predicción y si no podemos predecir del todo el comportamiento de un proceso y/o corregir la manera en que este puede generar situaciones concurrentes o asíncronas, para eso podemos conocer una manera de implementar funciones y utilizar las API's de un lenguaje como lo es Dart. Así que abordamos algunos de los conceptos teóricos de la concurrencia, la programación asíncrona, y algunos detalles de las herramientas que el lenguaje nos proporciona para implementar soluciones muy interesantes a cada situación. Al final se muestran algunas pruebas de implementación tanto con la Máquina Virtual de Dart, como con un DartPad que se encuentra disponible en línea.

## **Introducción.**

Se desarrollaron ampliamente mucho de los conceptos fundamentales de la programación asíncrona, además partimos de definir un poco de lo que es la concurrencia para los sistemas operativos como esto guarda una amplia relación con el multiprocesamiento. Y como podemos generar esquemas de programación asíncrona para así desarrollar los mecanismos necesarios para manejar los eventos por ejemplo en una interfaz de usuario.

Y aunque pareciera que esto es un tema moderno y que se emplea demasiado en relación funcionalidades para aplicaciones web, en realidad este concepto tiene vigente desde hace mucho tiempo. Ya que es una manera de optimizar el proceso de ejecución de un programa. Y aunque se han hecho ya algunos esfuerzos por lo tanto, no deja de ser un tema interesante que tiene interesantes implementaciones con el hardware y los modelos actuales.

Los planteamientos que sugiere la programación asíncrona van encaminados a como las llamadas al sistema pueden responder inmediatamente, pero ser procesadas después. En ese sentido los lenguajes de programación como Dart, realizan esfuerzos para incluir clases dentro de la API del lenguaje, que proporcionan algunos métodos y elementos, como una propuesta para desarrollar códigos dentro de este paradigma de eventos y aislamiento.

Por otra parte, si se requiere manejar problemas como los que genera la concurrencia, en este caso utilizamos una clase que Dart tiene destinada para cierto tipo de situaciones, que es la

clase Isolate, que nos proporciona subprocesos con memoria “propia”, para que se puedan realizar las operaciones en paralelo necesarias.

Al final planteamos algunos ejemplos relacionados con implementaciones de los Futures, los Streams y las palabras reservadas de await y async. Tales ejemplos son un esfuerzo por comprender mejor la sintaxis del lenguaje y usarla a nuestro favor, de tal forma que nos permita sacarle partida, no se pudieron representar las ejecuciones dado que en su mayoría dependen del tiempo y no tenían gran relevancia como se plantearon, pero inclusive se pueden probar en la página de dart para comprobar lo interesante de su funcionamiento. Son códigos muy cortos y se analizan, se pueden variar algunos elementos y parámetros para ir jugando con las funciones.

## **Marco teórico.**

La concurrencia no se refiere a dos o más eventos que ocurren a la vez si no a dos o más eventos cuyo orden es no determinista, esto es, eventos acerca de los cuales no se puede predecir el orden relativo en que ocurrirán.

Los problemas de concurrencia son muy difíciles de detectar y más aún de corregir. Es importante y mucho más efectivo realizar un buen diseño inicial de un programa concurrente en lugar de intentar arreglarlo cuando se detecta alguna falla.

Tener más de un procesador, no sólo no soluciona el problema, sino que lo empeora. ahora las operaciones de lectura o escritura pueden ejecutarse directamente en paralelo y aparecen nuevos problemas de coherencia de caché (Gunnar Wolf, 2015).

**Operación atómica.** Manipulación de datos que requiere la garantía de que se ejecutará como una sola unidad de ejecución. El efecto de que se retire el flujo no llevará a un comportamiento inconsistente.

**Condición de carrera.** Categoría de errores de programación que involucra a dos (o más) procesos que fallan al comunicarse su estado mutuo.

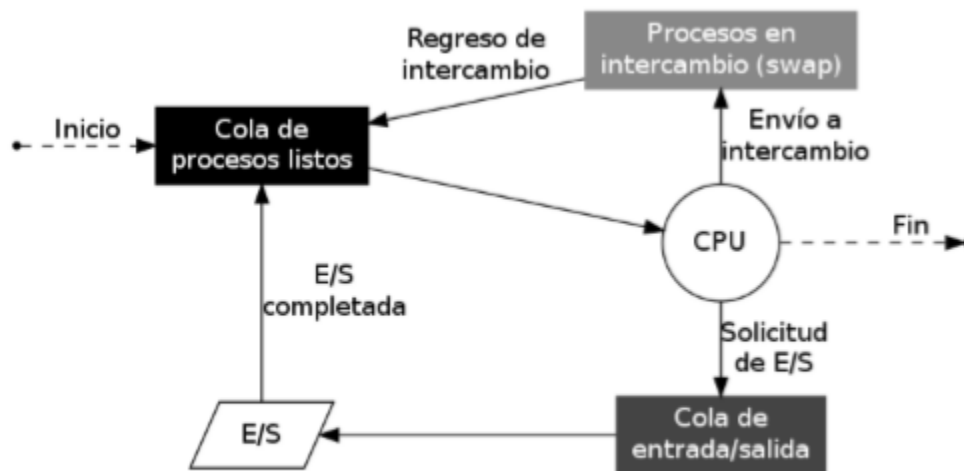
**Sección crítica.** En ella están los datos compartidos y por tanto requiere una protección especial ante los accesos simultáneos.

**Recurso compartido.** A este se puede tener acceso desde más de un proceso. en muchos escenarios esto es una variable de memoria, pero podrían ser archivos, periféricos, etcétera.

Conocer el fenómeno de la concurrencia nos acerca más a entender los sistemas operativos, ya que de esta forma entendemos conceptos que se generan a partir de como se administran procesos dentro de un espacio que a veces damos por sentado que está o debe estar presente en cualquier computadora en la que uno está trabajando. Sin embargo, quizá muchas veces

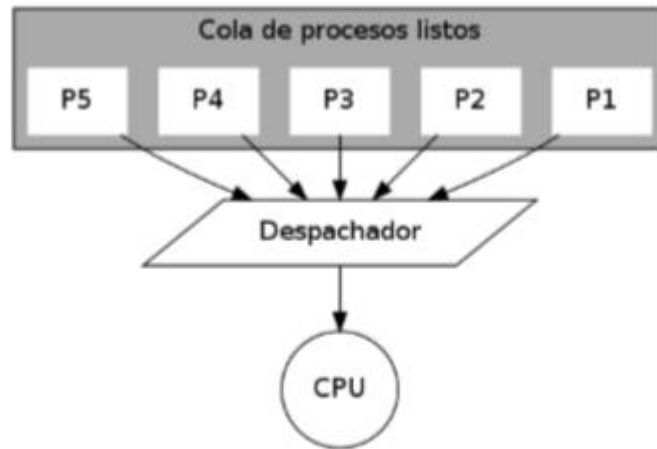
no somos conscientes de todo aquello que se tiene que gestionar desde el sistema operativo para que implementaciones basadas en paralelismo puedan funcionar y más aún que el hecho de comprender como se comportan algunas cosas como los hilos de procesamiento, pueden o no ayudar al rendimiento de nuestro programa, siempre y cuando sea prudente implementar nuestra solución a partir de los fundamentos teóricos.

En otro tenor, la programación asíncrona es un paradigma de programación que se utiliza para escribir aplicaciones que requieren múltiples hilos de procesamiento, así como también para integrar programas que están impulsados por eventos. En este paradigma los programas tienen la habilidad de hacer dos tipos de llamadas a funciones: las llamadas convencionales síncronas donde el sistema espera hasta que el llamado haya completado los cálculos necesarios, y por otro lado las llamadas asíncronas que regresan de inmediato, pero son procesadas después de haber sido enviadas por el planificador (Chadha, R. & Viswanathan, M., 2009).



*Figura 4.2: Planificador a mediano plazo, o agendador.*

Planificador a mediano plazo, o *agendador*. (Wolf, G., 2015)



**Figura 4.3:** Planificador a corto plazo, o *despachador*.

Planificador a corto plazo, o *despachador*. (Wolf, G., 2015)

Dichos sistemas se pueden abstraer convenientemente mediante autómatas con una pila para modelar el cálculo recursivo y una colección de llamadas asíncronas pendientes, con la restricción de que una nueva llamada síncrona se procese solo cuando la pila esté vacía.

La programación síncrona por otra parte se caracteriza por el hecho de que cada llamada que se escribió en el programa va en secuencia con la anterior; de este modo cada que se lanza una instrucción, esta será la única en ejecutarse durante el periodo de tiempo en que se realicen las operaciones necesarias para que la instrucción nos devuelva algún resultado. Así vemos que la programación síncrona podría implementarse a partir de uno o más hilos, pero en ambos casos el tiempo es “lineal”, en el sentido de que la secuencia de acciones esta encadenada a que el retorno de cada operación nos devuelva algún resultado. Por lo mismo el mínimo de tiempo al procesar con un hilo sería la suma de lo que tarda cada proceso (suponiendo que no haya interrupciones) y con más hilos tendríamos como mínimo lo que tarde el proceso más largo. Este último caso parece conveniente, mas existe la programación asíncrona que ya se ha mencionado y ahora se extienden los conceptos para su implementación.

Tipo de proceso	Cantidad de hilos: descripción	Imagen
Síncrono	1 hilo: el mínimo de tiempo es la suma del tiempo que tarde cada operación.	

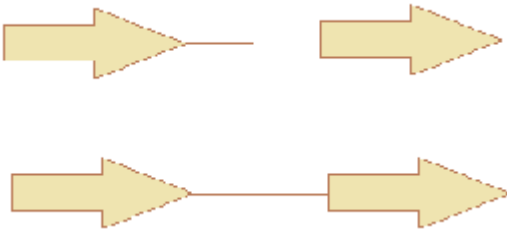
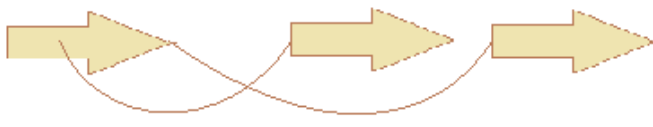
Síncrono	2 o más: el mínimo de tiempo es lo que tarda la operación que tarda más.	
Asíncrono	2 hilos o más: el tiempo “divide”, el proceso que inicio la acción puede seguir ejecutándose.	

Tabla 1. Principales tipos de procesos síncronos y asíncronos.

En ocasiones se suele ver la programación asíncrona como un esquema en donde varios eventos están ocurriendo al mismo tiempo. Sin embargo, esto podría parecer confuso dado que es más común que pensemos en un modelo de programación síncrona, estructurada y secuencial ¿Por qué?

Es posible que las nociones del proceso que realiza el sistema operativo para ejecutar un programa no sean del todo claras, como se le asignan los eventos a qué o a quién. ¿Qué es un evento? Resulta que los eventos son un conjunto de acciones que muy comúnmente se relacionan con las acciones que realiza un usuario sobre la interfaz en donde se ejecuta el programa de interés. En este sentido, el usuario puede presionar alguna tecla, mover el ratón o inclusive pueden ocurrir otras no tan visibles al usuario como transacciones de una base de datos. Todos estos eventos pueden estar manejados por un controlador de eventos, que su tarea básicamente es estar monitoreando si el usuario lleva a cabo alguna acción.

Parece no tener mucho sentido quizá de algún modo tener una estructura que se encargue de estar revisando solamente las acciones del usuario, pero es más ingenuo pensar que no se están llevando a cabo procesos cuando el usuario parece no estar haciendo nada. Resulta que esta estructura está siendo ejecutada (sobre todo en implementaciones modernas), por otro hilo que el sistema operativo designó para que precisamente el controlador o manejador de eventos se lo asigne a una cola de eventos, de esa cola se van descargando uno a uno, hasta vaciar la cola. En este punto la implementación teórica planteada en (Chadha, R. & Viswanathan, M., 2009), no resulta tan descabellada luego de que de verse a la programación asíncrona como un autómata de pila, ahora es en su implementación práctica una estructura que requiere de un medio de almacenamiento para ir guardando los eventos que se van generando a partir de la interacción del usuario.

En algunos lenguajes como javascript o python, se utilizan implementaciones a partir de “promesas” que encapsulan de cierta manera las funciones asíncronas del programa, de tal forma que el código se mantiene altamente legible, de tal forma que con ello sea más sencillo reducir los errores y que la corrección de estos mismo sea más sencilla.

Entonces, tenemos por un lado los programas de tipo síncronos que parecen lucir como implementaciones estructuradas y secuenciales, mas es importante decir que para este punto las acciones que lleva a cabo el lenguaje que a continuación se desarrollará, hace que este encapsulamiento sea un tanto más refinado y que se pueda partir algunas veces de un código síncrono que amarrando algunos detalles se llega a convertir en asíncrono.

¿Y qué tiene que ver todo esto con el sistema operativo? Bueno en un principio se podría decir que el hecho de tener una cola de eventos y algo que vaya agregando los eventos la cola es una gran solución y que nos facilita mucho trabajar mediante programación asíncrona. Sin embargo, algunos procedimientos de los que se encarga el sistema operativo para que esto funcione adecuadamente, resulta que se vuelven más complejos cuando nos damos cuenta de que en realidad las computadoras de hoy en día pueden contar con varios núcleos y por lo tanto la paralelización que se puede realizar para trabajar funcionalidades asíncronas resulta también muy compleja.

Si el sistema operativo funciona bajo principios preventivos, que sería lo más probable en este caso de programación asíncrona, tenemos entonces lo siguiente:

La distinción más importante es que en un sistema preventivo el programador no tiene que preocuparse por mantener ocupado el hilo principal. En unos pocos milisegundos, el hardware interrumpe el hilo en ejecución y comienza a ejecutarlo el sistema operativo. El sistema operativo elige otro proceso para iniciarlo y lo ejecuta. En un sistema preventivo, el hilo principal se comparte automáticamente de manera justa entre todos los procesos, sin importar cómo estén programados. (James, M., 2013).

Así vamos a tener algunos detalles para la implementación de funciones asíncronas que podrían ser graves como el hecho de tener una operación muy larga o que no termine y en ese caso, el modelo de programación asíncrona se vuelve algo obsoleto, pues tendríamos al usuario esperando a que el proceso respondiera, sin que eso vaya a suceder.

Existen algunos conceptos que es importante definir para entender mejor como se implementan el modelo asíncrono en el lenguaje de programación, Dart:

**Operación síncrona:** una operación síncrona bloquea la ejecución de otras operaciones hasta que se completa.

**Función síncrona:** una función síncrona sólo realiza operaciones síncronas.

**Operación asincrónica:** una vez iniciada, una operación asincrónica permite que otras operaciones se ejecuten antes de que se complete.

**Función asincrónica:** una función asincrónica realiza al menos una operación asincrónica y también puede realizar operaciones sincrónicas.

Una vez definidos la mayor parte de los conceptos teóricos, se procede a desarrollar algunos conceptos ya propiamente del lenguaje de programación Dart, de forma que nos permita llegar a entender un ejemplo de implementación un tanto mejor.

## **Desarrollo.**

A continuación, se describen algunas características del lenguaje, para después abordar su sintaxis y posteriormente los ejemplos de implementación para concurrencia y asincronía.

## **Historia.**

Dart fue presentado en la conferencia GOTO en Aarhus, Dinamarca, del 10 al 12 de octubre de 2011. El proyecto fue fundado por Lars Bak y Kasper Lund. Dart 1.0 fue lanzado el 14 de noviembre de 2013. En agosto de 2018, se lanzó Dart 2.0, con cambios de idioma que incluyen un sistema de tipo de sonido.

Recientemente, el lanzamiento de Dart 2.6 se acompaña con una nueva extensión dart2native. La característica extiende la compilación nativa a las plataformas de escritorio Linux, macOS y Windows. Los desarrolladores anteriores podían crear nuevas herramientas solo con dispositivos Android o iOS. Además, con esta extensión es posible componer un programa Dart en ejecutables autónomos. Por lo tanto, según los representantes de la compañía, ahora no es obligatorio tener instalado Dart SDK, los ejecutables autónomos ahora pueden comenzar a ejecutarse en unos segundos. La nueva extensión también está integrada con el kit de herramientas Flutter, lo que permite utilizar el compilador en servicios pequeños, como soporte de backend, por ejemplo. (Wikipedia, 2020).

## **Estandarización**

Ecma International ha formado el comité técnico TC52 para trabajar en la estandarización de Dart, y dado que Dart puede compilarse en JavaScript estándar, funciona de manera efectiva en cualquier navegador moderno. Ecma International aprobó la primera edición de la especificación del lenguaje Dart en julio de 2014, en su 107a Asamblea General, y una segunda edición en diciembre de 2014. La última especificación está disponible en la especificación del lenguaje Dart. (Wikipedia, 2020).

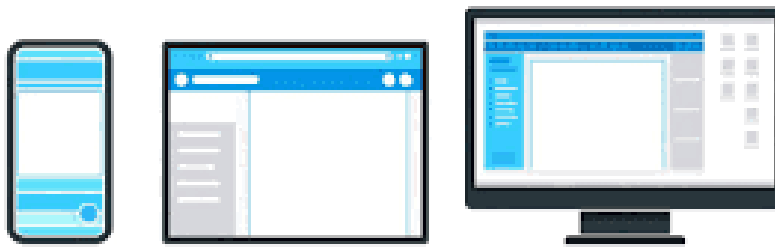
## **Dart y Flutter.**

Flutter es un framework multiplataforma que pretende llevar el desarrollo de aplicaciones móviles nativas, desde una misma base de código, hasta distintos sistemas operativos como iOS y Android. Utiliza un lenguaje de programación basado en javascript, el cual permite la implementación de distintos recursos; llamado dart.





Logo de Dart (Wikipedia, 2020).



Entorno de desarrollo, *Flutter* (Wikipedia, 2020).

Dart es un lenguaje de programación web, utiliza el paradigma orientado a objetos. Utiliza una máquina virtual como Java, esto quiere decir que puede lograr gran compatibilidad (como con Java). La máquina virtual de Dart se llama Dart VM.

### **Concurrencia en Dart.**

La condición de carrera puede causar serios errores, incluyendo pérdida de datos. La forma típica de arreglar una condición de carrera es proteger el recurso compartido mediante un bloqueo que impide la ejecución de otros subprocesos, pero los bloqueos pueden traer problemas como *deadlock* y *starvation*.

### **La clase *Isolate*, subprocesos.**

Dart tomó un enfoque diferente a este problema. Los subprocesos de Dart, llamados *isolates*, no comparten memoria, lo que evita la necesidad de realizar una gran cantidad de bloqueos. Los *isolates* se comunican pasando mensajes a través de canales.

Un aislamiento es aquello que al ejecutarse el código Dart; es como un pequeño espacio en la máquina con su propia porción privada de memoria y un solo hilo que ejecuta un bucle de eventos. En muchos otros lenguajes como C ++, puede tener varios subprocesos que comparten la misma memoria y ejecutan el código que desee. Sin embargo, en Dart, cada subproceso está aislado con su propia memoria, y el subproceso solo procesa eventos.

Muchas aplicaciones Dart ejecutan todo su código en un solo *Isolate*, pero puede tener más de uno si lo necesita. Esto se hace con una función. `spawn()`, la cual crea un aislamiento por separado para hacer las operaciones que requiera, dejando su aislamiento principal libre para, por ejemplo, reconstruir y renderizar el árbol de widgets.

La asignación de memoria y la recolección de basura en un aislamiento no requieren bloqueo. Solo hay un hilo, así que, si no está ocupado, sabes que la memoria no está siendo mutada. Eso funciona bien para las aplicaciones Flutter, que a veces necesitan acumularse y destruir un montón de widgets rápidamente.

Dart, al igual que JavaScript, es single thread, lo que significa que no permite la prioridad en absoluto. En su lugar, los subprocesos ceden explícitamente (usando `async/await`, Futures, o Streams). Esto le da al desarrollador más control sobre la ejecución. Un único subproceso ayuda al desarrollador a garantizar que las funciones críticas (incluidas las animaciones y las transiciones) se ejecuten hasta su finalización, sin necesidad de un preaviso. Esto es a menudo una gran ventaja no sólo para las interfaces de usuario, sino también para otros códigos cliente-servidor.

Por supuesto, si el desarrollador olvida ceder el control, esto puede retrasar la ejecución de otro código. Sin embargo, hemos descubierto que es mucho más fácil encontrar y arreglar el olvido del rendimiento que el olvido del bloqueo (porque las condiciones de la competición son difíciles de encontrar).

### **¿Eventos que aún no han ocurrido? Future.**

Un Future es lo mismo que una promesa y un Stream es como un Future recurrente. Por ejemplo, un `Future<String>` es un objeto que representa que, en algún momento, una computación asíncrona devolverá un String, mientras que un `Stream<String>` es un objeto que representa que una computación asíncrona generará uno o más Strings de forma recurrente en el futuro. (Zaera, I., 2013).

Es decir, cuando hablamos de programación asíncrona, en la sintaxis de Dart existe una cierta facilidad para declarar los objetos y los retornos de estos que ya se identifican propiamente como programación asíncrona. En javascript es mucho más común ver las promesas, como objetos que nos devuelven estados posibles para las operaciones que se están realizando de manera asincrónica. Mas en Dart se existe esto mismo, sumándole los Streams, que son como bandas de transporte que hacen que estas promesas sean recurrentes, o visto de otro modo que se vayan entregando el estado de cada promesa con cierta frecuencia de manera iterativa.

### **Keywords `async` y `await`.**

Existen dos palabras que ayudan a convertir de manera muy sutil e inclusive en ocasiones elegante la sintaxis de código asíncrono en Dart. Estas simples palabras transforman por completo la representación del código, pues en lugar de hacer que se tengan que ir encadenando las llamadas para representar la asincronicidad como en javascript, en Dart sucede que algo que parece estar escrito síncrona y estructuradamente, se puede modularizar

más cómodamente con estas palabras reservadas que también nos ayudan a definir las características de las funciones y que se van a ir completando con Futures o Streams.

**async.** puede usar la palabra clave `async` antes del cuerpo de una función para marcarla como asíncrona.

**async function.** una función asíncrona es una función etiquetada con la palabra clave asíncrona.

**await.** puede usar la palabra clave `await` para obtener el resultado completo de una expresión asincrónica. La palabra clave `await` solo funciona dentro de una función asíncrona.

## Ejemplos de implementación.

Los ejemplos de implementación fueron escogidos de los códigos ejemplo disponibles en la página de Dart, algunos se decidieron modificar para tener otras posibilidades. Sin embargo, actualmente se sigue experimentando con tales posibilidades, a fin de descubrir un ejemplo más interesante.

```
Future<int> esperaTarea() async {  
  await Future.delayed(const Duration(seconds: 5));  
  return 0;  
}  
  
main () async {  
  int result = await esperaTarea();  
  print(result);  
}
```

Se prueba el uso de Future, para esperar una cuenta y luego de eso se imprime la acción pendiente que es el retorno de la función. Código recreado tomando como fuente el sitio oficial de Dart: <https://dart.dev/codelabs/async-await>

```

Stream<int> count() async* {
  for(int i=1; i <= 15; i++) {
    await sleep1();
    yield i;
  }
  for(int i=1; i <= 10; i++) {
    await sleep2();
    yield i;
  }
}

Future sleep1() {
  return new Future.delayed(const Duration(seconds: 1), () => "1");
}

Future sleep2() {
  return new Future.delayed(const Duration(seconds: 2), () => "2");
}

main() async {
  await for (int i in count()) {
    print(i);
  }
}

```

▶ RUN

Se crearon dos future para que cada uno lleve una cuenta a distintos intervalos, la función asíncrona de count, va recibiendo los resultados en un Stream y se los va proporcionando poco a poco a el await del main, de tal forma que primero se hace una cuenta y luego otra, pero lo interesante aquí es que es esta es la azúcar sintáctica de dart.

```
import 'dart:async';

Future<int> sumStream(Stream<int> stream) async {
  var sum = 0;
  await for (var value in stream) {
    sum += value;
  }
  return sum;
}

Stream<int> countStream(int to) async* {
  for (int i = 1; i <= to; i++) {
    yield i;
  }
}

main() async {
  var stream = countStream(100);
  var sum = await sumStream(stream);
  print(sum); // 5050
}
```

Se hace la suma total de un Stream de enteros, usando asincronía nuevamente, por lo que la variable de suma espera a ser inicializada para entregar sus resultados. *Código recreado tomando como fuente el sitio oficial de Dart: <https://dart.dev/codelabs/async-await>*

```
import 'dart:isolate';

void foo(var message){
  print('execution from foo ... the message is :'+ message);
}

void main(){
  Isolate.spawn(foo, 'Hello!!');
  Isolate.spawn(foo, 'Greetings!!');
  Isolate.spawn(foo, 'Welcome!!');

  print('execution from main1');
  print('execution from main2');
  print('execution from main3');
}
```

Manejo de la concurrencia con dart, creando regiones aisladas.

```
C:\Users\niver\Documents\expo_dart>dart concurrency.dart
execution from main1
execution from main2
execution from main3
execution from foo ... the message is :Greetings!!
execution from foo ... the message is :Hello!!
```

Esta salida si tiene interpretación. Las de programación asíncrona dependen del tiempo.

```
import 'dart:async';

Future<int> sumStream(Stream<int> stream) async {
  var sum = 0;
  try {
    await for (var value in stream) {
      sum += value;
    }
  } catch (e) {
    return -1;
  }
  return sum;
}

Stream<int> countStream(int to) async* {
  for (int i = 1; i <= to; i++) {
    if (i == 4) {
      throw new Exception('Intentional exception');
    } else {
      yield i;
    }
  }
}

main() async {
  var stream = countStream(10);
  var sum = await sumStream(stream);
  print(sum); // -1
}
```

Es la misma suma de números, pero ahora se atrapan excepciones específicas de manera asíncrona tanto en el Stream, como en el Future. Código recreado tomando como fuente el sitio oficial de Dart: <https://dart.dev/codelabs/async-await>

## Conclusión.

La implementación de funciones asíncronas y concurrencia en Dart implica mucha disposición a pasar un rato revisando la tecnología, pues ya que resulta muy similar a javascript, es cómodo acostumbrarse a sus sintaxis. Algunos detalles respecto a la implementación de los eventos Futuros, que son muy similares a las promesas de javascript, tienden a ser colocados en justa medida como una herramienta muy útil que, de la mano con los Streams, logra ofrecernos bastantes posibilidades de desarrollo.

La inmensa gama de recursos con que cuenta Dart nos permite hacer la mayor parte de distintos paradigmas, de tal forma que suceden muy bien algunas cosas de manejo de eventos. Si se puede emplear este lenguaje para marcos de desarrollo que impliquen más allá de las aplicaciones móviles, no sé si logre desplazar en algún punto los lenguajes como java y Codeline que aún están en el mercado ofreciendo igual, manejo de programación asíncrona y cosas similares, pero que no se unifican en un marco de trabajo como lo hace Dart. Por otro lado, si Google, no se apodera de estos recursos en algún momento o pasa otra circunstancia adversa que no deje seguir desarrollando aplicaciones dentro de tal marco de trabajo y con este lenguaje, puede que comience a emerger en poco tiempo.

## Referencias.

- Wolf, G., Ruiz, E., Bergero, F. & Meza, E. (2015). *Fundamentos de sistemas operativos*. México, D.F: Universidad Nacional Autónoma de México.
- Chadha, R. & Viswanathan, M. (2009). *Deciding branching time properties for asynchronous programs*. marzo 08, 2020, Recuperado de Elsevier Sitio web: <https://reader.elsevier.com/reader/sd/pii/S0304397509000851?token=8F59EB69369C3057261836611C3E1D9D914FA6F82B6968E7B972EFE904B40DDB893D0D2665976EE470965DF004874FDB>
- Haverbeke, M. (2018). *Eloquent JavaScript* 3ra edición. marzo 12, 2020. Recuperado de del recurso electrónico: <https://eloquentjs-es.theddojo.mx/>. Cap. 11. Programación asíncrona: [https://eloquentjs-es.theddojo.mx/11\\_async.html](https://eloquentjs-es.theddojo.mx/11_async.html)
- tutorialspoint. *Dart Programming - Concurrency*. marzo 02, 2020. Recuperado de: [https://www.tutorialspoint.com/dart\\_programming/dart\\_programming\\_concurrency.htm](https://www.tutorialspoint.com/dart_programming/dart_programming_concurrency.htm)
- Wikipedia(2020). *Dart(programming language)*. marzo 01, 2020. Recuperado de: [https://en.wikipedia.org/wiki/Dart\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Dart_(programming_language))
- Dart. *Isolate class*. marzo 02, 2020. Recuperado de: <https://api.dart.dev/stable/2.7.1/dart-isolate/Isolate-class.html>
- stackoverflow. *What's the difference between async and async\* in Dart?*. marzo 13, 2020. Recuperado de: <https://stackoverflow.com/questions/55397023/whats-the-difference-between-async-and-async-in-dart>
- Dart. *Asynchronous programming: futures, async, await*. marzo 02, 2020. Recuperado de: <https://dart.dev/codelabs/async-await#example-async-and-await-with-try-catch>

- Wikipedia(2008). *Planificador*. marzo 05, 2020. Recuperado de: <https://es.wikipedia.org/wiki/Planificador>
- Walrath, K (2019). *Dart asynchronous programming: Isolates and event loops*. marzo 05, 2020. Recuperado de: <https://medium.com/dartlang/dart-asynchronous-programming-isolates-and-event-loops-bffc3e296a6a>
- Appdelante. (2017). *Asincronía en JavaScript - Parte 1 - Sincronía y Concurrencia*. marzo 14, 2020. Recuperado de: <https://www.youtube.com/watch?v=PndHsDpEfh>
- Flutter(2019). *Dart Streams - Flutter in Focus*. marzo 13, 2020. Recuperado de: <https://www.youtube.com/watch?v=nQBpOIHE4eE>
- Flutter(2019). *Async/Await - Flutter in Focus*. marzo 13, 2020. Recuperado de: <https://www.youtube.com/watch?v=SmTCmDMi4BY>
- Zaera, I. (2013). *Programación Asíncrona en Dart: Funciones y Streams*. Marzo 18, 2020. Recuperado de: <http://blog.agilar.org/2013/04/05/programacion-asincrona-en-dart-funciones-y-streams/>
- Mike, J. (2013). *What Is Asynchronous Programming?* i-Programmer. Marzo 14, 2020. Recuperado de: <https://www.i-programmer.info/programming/theory/6040-what-is-asynchronous-programming.html>
- Mike, J. (2013). *Managing Asynchronous Code - Callbacks, Promises & Async/Await*. i-Programmer. Marzo 14, 2020. Recuperado de: <https://www.i-programmer.info/programming/theory/6040-what-is-asynchronous-programming.html>